Johannes Feichtner, BSc

CryptoSlice

Static Analysis of Cryptography in Android Applications

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

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

Institute for Applied Information Processing and Communications (IAIK)

Graz, May 2015

AFFIDAVIT

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 indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

SIGNATURINFORMATIONEN	**Unterzeichner**	Johannes Ferdinand Feichtner
	Datum/Zeit-UTC	2015-05-04T16:24:27+02:00
	Prüfinformation	Informationen zur Prüfung der elektronischen Signatur finden Sie unter: https://www.signaturpruefung.gv.at
Hinweis	Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument ist gemäß § 4 Abs. 1 Signaturgesetz einem handschriftlich unterschriebenen Dokument grundsätzlich rechtlich gleichgestellt.	

Date

Signature

Abstract

As mobile platforms enjoy great popularity, software vendors use mobile devices also to deploy applications which process sensitive data. Unfortunately, they often provide only little details on how responsibly critical information, such as passwords and encryption keys, are protected. Users can hardly be sure whether programs handle entered credentials accordingly. A wrong application of cryptography or security-critical APIs, however, may expose secrets to unrelated parties and, thereby, undermine the intended security level.

A wide range of Android applications perform sensitive tasks and, therefore, rely on protective mechanisms that are natively provided by the operating system. Manually verifying the secure implementation of critical functionality can be challenging due to the rising complexity and size of todays programs. Automated alternatives, on the other side, are usually targeted to uncover general security problems, rather than evaluating particular properties.

In this thesis, we introduce a new static analysis framework, named CryptoSlice, which excels in identifying and analyzing security-related code in Android applications. The modular architecture of the Java-based framework offers the ability to automatically extract, disassemble and investigate programs. By using static backtracking on definable slicing patterns, the information flow of relevant code segments is opened up and can be further inspected. For this purpose, we include a set of security rules, designed to highlight the improper usage of security-relevant functionality. As a result, we are not only able to confirm the existence of problematic statements but can also pinpoint their exact origin and find appendant invocations.

CryptoSlice pursues a novel approach to follow the trace of a password right from the point where it enters an application. Starting at a definable trigger point, such as an input field, we can statically compute all possible data flows in forward direction. The subsequently conducted analysis of the result is suited to reveal potential security problems and data leaks.

By applying our security rules manually and automated on a set of applications that process sensitive data, we evaluate their accuracy and gain a valuable insight into the prevalence of security-critical misconceptions in current Android applications.

Keywords: Android, Security, Static Analysis, Slicing, Cryptography, Passwords

Kurzfassung

Anlässlich der großen Beliebtheit mobiler Plattformen werden auf entsprechenden Geräten auch Applikationen eingesetzt, die sensible Daten verarbeiten. Bedauerlicherweise sind von Herstellern oft nur wenig Details dazu bekannt, auf welche Art und Weise heikle Informationen wie etwa Passwörter oder kryptographische Schlüssel geschützt werden. Anwender können kaum abschätzen ob es ungefährlich ist, persönliche Daten in eine Applikation einzugeben. Eine falsche Anwendung von Kryptographie oder sicherheitskritischen Schnittstellen kann begünstigen, dass Unberechtigte an sensible Daten gelangen oder die eigentlich beabsichtigte Sicherheit nicht gegeben ist.

Eine große Anzahl von Android-Applikationen führt sicherheitskritische Aufgaben aus und greift dafür auf Schutzmechanismen zurück, die vom System bereitgestellt werden. Die manuelle Überprüfung ob diese Funktionalität sicher implementiert ist, kann durch steigende Komplexität und Größe moderner Applikationen deutlich erschwert werden. Automatisierte Alternativen sind oft daraus ausgerichtet, allgemeine Sicherheitsprobleme aufzuzeigen, können aber häufig keine spezifischen Sicherheitseigenschaften evaluieren.

In dieser Arbeit stellen wir CryptoSlice vor, ein Werkzeug für statische Analyse, das sich beim Auffinden und Auswerten von sicherheitskritischem Code in Android-Applikationen auszeichnet. Die modular-zusammengesetzte Architektur des Java-basierten Werkzeugs hilft bei der automatisierten Verarbeitung und Inspektion von Programmen. Durch statisches Backtracking von festlegbaren Mustern lässt sich der zugehörige Informationsfluss nachvollziehen und kann weiter untersucht werden. Zu diesem Zweck stellen wir Regeln bereit, die die unsachgemäße Verwendung von sicherheitsrelevanten Komponenten aufdecken. Es ist uns letztlich nicht nur möglich, die reine Existenz von Problemen zu bestätigen sondern sogar deren Herkunft exakt aufzuzeigen.

CryptoSlice verfolgt einen innovativen Ansatz um den Datenfluss eines Passworts von dem Moment an zu verfolgen, wo es in eine Applikation eingegeben wird. Beginnend bei einem definierbaren Startpunkt, wie etwa einem Eingabefeld, kann statisch jeder mögliche Informationsfluss in Vorwärtsrichtung erhoben werden. Die darauffolgende Analyse eignet sich dazu, potentielle Sicherheitsprobleme und Datenlecks sichtbar zu machen.

Durch die manuelle und automatisierte Anwendung der Regeln auf Test-Applikationen, die sensible Daten verarbeiten, evaluieren wir deren Genauigkeit und erhalten einen Einblick in die Verbreitung von sicherheitsgefährdenden Fehlern in aktuellen Android-Applikationen.

Schlüsselwörter: Android, Sicherheit, Statische Analyse, Slicing, Kryptographie, Passwörter

Acknowledgements

First, I would like to express my gratitude to my advisors Peter Teufl and Daniel Hein for their professional guidance and ceaseless support. Their continuous encouragement and precious feedback fostered my enthusiasm for this work.

I'm also thankful for the stimulating and challenging environment at IAIK where I was given the opportunity to work on this exciting subject. Numerous discussions with colleagues and friends motivated me to stay focussed and finish my studies. Furthermore, the LaTeX skeleton of Keith Andrews facilitated writing this thesis.

Finally, I especially wish to thank my wife and family for their comprehensive support and valuable advice. Their generous love and motivating words supported me with confidence.

Johannes Feichtner

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Smartphones are ubiquitous in our daily lives and facilitate mobile working. Extensive technological capabilities of these devices enable consumers to carry out tasks that would have required a personal computer some years ago. Since the Android operating system is among the most popular mobile platforms, many software developers are also deploying programs that process sensitive user data. A commonly used method to protect this information, is the use of security APIs and cryptographic functionality, provided by the platform. Manually inspecting whether the mechanisms have been called the way they are designed for can be a challenging endeavour. The rising complexity and sophisticated code obfuscated techniques impede a conclusive security analysis of security-critical programs. Automated analysis tools, in contrast, are often powerful in general but fail to evaluate specific security parameters.

As a remedy, we have developed CryptoSlice, a framework to identify and analyze security-related code in Android applications. It includes an integrated analysis environment which automatically carries out the inspection of a program. By producing well-arranged data flow graphs for arbitrary statements it facilitates the manual inspection of applications. Using a set of security rules, CryptoSlice is also capable of automatically identifying common implementation flaws and tracks down wrongly chosen security attributes.

In the following, we first point out the relevance of a targeted application analysis by referring to disclosed insufficiencies in popular programs. Subsequently, we introduce our static analysis framework CryptoSlice and highlight its capabilities.

1.1 Protecting Sensitive Data on Android

A wide range of Android applications perform security-critical tasks and require that user inputs are processed reliably. For this to achieve, a correct implementation is indispensable, ensuring that no sensitive data can be leaked within the data flow and that cryptographic systems are applied correctly, in case their use is appropriate. Sadly, there is little information on how responsibly applications treat critical user inputs, such as passwords. Usually it is unknown whether a program applies cryptography correctly and if it is safe for a user to enter sensitive data.

For example, when Mobile Banking applications access a customer's bank account, they ideally establish a connection to the bank by calling a HTTPS-secured web address. Subsequently, previously entered login credentials are sent to the bank. The problem, however, is that even in this perfectly valid data flow example, customer login data is prone to be stolen if the application implementation fails to correctly validate the authenticity of the remote party. In case of SSL/TLS

connections, apps must not accept arbitrary, probably expired, self-signed, or untrusted certificates in order to prevent Man-in-the-Middle (MITM) attacks. According to Fahl et al. [14], 24 of 43 tested Mobile Banking applications were unable to withstand the performed MITM attack. Similarly, consumers are expecting password manager applications to safely guard their personal login data. By using phrases, such as "military-grade encryption" or "ultra-secure protection", software vendors, on the other hand, inspire confidence among consumers that the product excels with outstanding security. After analysing more than 11.000 Android applications that use cryptographic APIs, the empirical study of Egele et al. [9] concludes that 88% commit at least one precarious implementation mistake. A case study of Belenko and Sklyarov [6] demonstrates that very similar issues are also to be encountered with applications, designed for Apple iOS or BlackBerry. However, due to different attack vectors on each mobile platform, the actual risk potential of a specific vulnerability is not equal for all operating systems but needs to be assessed individually.

The analysis of applications is basically done by performing static or dynamic analysis. Depending on the threat model and platform specifics, one or even both approaches can be pursued for analysis purposes. In case of static analysis, a predefined set of inspection procedures is applied to an application. Thereby, although the program is never executed and regardless of the actual behaviour, it is possible to draw conclusions based on selected features. Dynamic analysis, in contrast, runs the application in a protected environment and tries to recognize previously defined patterns by means of interaction. A major advantage of static analysis involves the possible coverage of application code. For example, when analysing the data flow of password inputs, dynamic analysis would only match those password fields that were actually visited during execution. Static analysis, on the other hand, is instantly applicable to the totality of password fields, contained within an application.

1.2 Uncovering Critical Problems with CryptoSlice

In this thesis, we introduce a new static analysis framework, named CryptoSlice, which is capable of identifying and analysing security-critical code in Android applications. The Java-based framework offers the ability to automatically extract, disassemble and investigate applications. By applying static backtracking techniques, the control and data flow of relevant code segments is opened up and serves as input for further evaluation. For this purpose, the framework incorporates rules to make reliable assumptions about the degree of security, common cryptography-related constructions are able to provide. Apart from analysing these constructions, it is determined whether they deviate from being employed correctly. The framework, furthermore, supports the manual analysis of applications by delivering well-arranged graphs, representing the static slice of user-definable patterns. Due to the modular construction, basically any static analysis approach could be easily added to the sophisticated analysis environment.

To our knowledge, we are the first being able to follow the trace of a password throughout an Android application. The implemented forward slicing technique is used to track all occurrences of password fields, while simultaneously mapping the visited code lines to a graph representation. Subsequently, the graph is automatically examined involving predefined rules, in order to detect potential security problems. For this to achieve in a most accurate way, backward slicing enables the discovery of influencing parameters. Having determined all relevant variables, the mined context is inspected regarding cryptography-related misconceptions and possible data leak. Finally, it is possible to grade the attained security level, which can be achieve when a password is entered by a consumer.

Aside from tracking password fields, CryptoSlice is intended to identify and analyse security-

critical code. Therefore, a set of rules has been implemented whereas each rule first tries to find predefined method signatures and then, based on the found matches, verifies if matches the predefined standard model. In case a rule is violated, an alert is issued, accompanied by details about the problematic statement. Thereby, the framework manages to evaluate the exact location of rule violating code lines and writes information about the involved class, method and code line to the console as well as to a XML report. As a consequence, the result is easily readable and can moreover be interpreted automatically by parsing the XML report. Although Android code is usually based on Java-written code, we do not try to decompile Dalvik bytecode back to Java. As pointed out by Enck and Octeau [10], the creators of the ded decompiler, decompilation is only likely to succeed for 95% of the application classes. Instead, we disassemble Dalvik bytecode to Smali code, a register-based intermediate language. This delivers an output, similar to Java code but keeps it employable to our light-weight static slicing techniques. The chosen slicing algorithms are not contingent on previously generated program dependency graphs and, hence, support fast and computationally inexpensive program analysis. Nevertheless, in order to leverage the potential of today's multi-core processors, the framework supports the analysis of multiple Android applications in parallel. Thereby, it is possible to investigate a large amount of applications in minimal time and quickly identify applications that need manual analysis.

The solid framework and implementation of reliably working slicing algorithms support the detection of security-critical code. Due to our particular analysis approach, we excel in identifying problematic statements and appendant invocations. So, in contrast to the work of Egele et al. [9], we can give a clear advice about the origin of rule-violating code, rather than being only able to confirm its existence. In this thesis, we elaborate the used rules and apply them to a set of selected applications in order to verify their accuracy. In a subsequent case study, we highlight commonly encountered problems and refer to possible reasons. Due to the fact that many applications use third-party libraries containing cryptography-related code, our framework incorporates measures to evade over-counting of problematic code. Therefore, we hold a list of known libraries, mostly used for statistics or advertisement. Similarly, CryptoSlice supports the detection of advertisement-related code in general and can be instructed to do filtering at an early stage. As a result, only code objects which are relevant for the analysis are held in memory.

Although the provided cryptography-related rules enable a profound program inspection, it seems reasonable to extend the framework in future work, to cover more security-critical mistakes. Due to a very flexible and well-arranged framework design, new rules and detection features can easily be added. Additionally, the framework contains a heuristic search feature, which allows a quick evaluation of promising search keys.

1.3 Outline

The following chapters of this thesis are structured as follows:

Chapter 2 provides an overview of the theory, this thesis is based on. First, the Android platform architecture is introduced by presenting the main elements and fundamental security concepts. Subsequently, the internal structure of Android applications is explained. We address the underlying application components and clarify their use within the Android framework. Next, we introduce the Dalvik virtual machine, which represents the protected environment in which Android applications are run. After briefly denoting Dalvik bytecode, we pass on to an introduction of the Smali language, which basically is a higher-level abstraction of Dalvik bytecode. Using Smali code as input, the CryptoSlice tool performs static slicing, which is described in the subsequent section. Thereby, we distinguish between static and dynamic slicing and explain approaches

for forward and backward slicing. Besides, we induce control flow graphs which form the basis for the slicing graphs, generated by CryptoSlice. Furthermore, we highlight the cryptographic principles, which have been applied for this thesis. We shortly define symmetric and asymmetric cryptography and proceed with the concept of hash functions. In the following, we also illuminate the generation of random numbers in a Java-based environment. Equally related to the used security-critical rules, the purpose of password-based key derivation functions is explained, followed by particularities when employing cryptography on Android-based devices. Finally, the possibilities for secure storage of data on Android are elaborated.

Chapter 3 discusses related work. We list similar approaches for static analysis of Android applications and denote the applied concepts. Moreover, research regarding the security of Android applications is considered as well as studies concerning Java-based cryptography and password-based key derivation functions, in particular.

In Chapter 4, the CryptoSlice tool is introduced and the main aspects are first regarded from an external point of view. We highlight the overall capabilities and explain the technical surrounding, the framework is embedded in. The typical analysis workflow is modelled and the included steps are investigated in detail. Commencing with input preprocessing, we explain the internal representation of an application, followed by a description of the consecutive analysis steps and the finalising result reporting.

Chapter 5 focusses on static slicing of Smali code. We demonstrate the use of the slicing criterion and explain how it is processed in order to start slicing. Related to this, the tracking of password fields receives special attention as the slicing criterion needs to be determined dynamically, involving appropriate preprocessing. Next, the implemented techniques for forward and backward slicing of Smali code are illustrated. In addition, we describe the construction of a slicing graph, used to support manual analysis and to visualise the inter-procedural data flows. At last, we disclose the limitations of the chosen approaches.

In Chapter 6, we elaborate the implemented security-critical rules. For each rule, we make a problem statement and assess the risk exposure. Code examples are provided in order to illustrate wrong applications of security-critical functionality. Finally, we explain the chosen strategies for the detection of misuse.

The results of a framework evaluation are provided in Chapter 7. By performing manual and automated analysis on a set of applications, we evaluate the accuracy of the implemented security rules. We, furthermore, gain an insight into the distribution of security-related misconceptions in Android applications.

Chapter 8 concludes this work. We summarise the thesis and point out the valuable contribution, CryptoSlice is able to provide. Finally, we give an outlook and propose related subjects that could be investigated in future work.

Chapter 2

Background

In this chapter, we present necessary background information on the topic of this thesis. Section 2.1 introduces the Android platform architecture and discusses vital elements. By explaining fundamental security concepts, Android's security design is elaborated and an overview about the basic protection mechanisms is given. Subsequently, in Section 2.2 the internal structure of Android applications is examined. We address the underlying application components and clarify their use within the Android framework.

Next, in Section 2.3 we introduce the Dalvik virtual machine, which represents the protected environment in which Android applications are run. After briefly illustrating the translation from Java classes to Dalvik bytecode, the internal structure of Dalvik executable files is elaborated. In Section 2.4 we pass on to an introduction of the Smali language, which basically is a disassembled representation of Dalvik bytecode. Section 2.5 outlines the process of decompiling Android applications and lists the most popular tools.

Using Smali code as input, our framework performs slicing, which is described in Section 2.6. First, we distinguish between static and dynamic slicing. Second, we explain the general approaches for forward and backward slicing in Section 2.6.4. Besides, in Section 2.6.5 we induce data flow graphs which form the basis for the slicing graphs, generated by CryptoSlice.

Finally, we highlight the cryptographic principles, which have been applied for this thesis. We briefly explain symmetric cryptography in Section 2.7.2, and the available modes of operation as well as asymmetric cryptography in Section 2.7.3. We proceed with the concept of hash functions in Section 2.7.4 and illustrate the designated usage. In the following, we also explain the generation of random numbers in a Java-based environment and show the consequences of incorrect application. Equally related to the used security-critical rules, the purpose of password-based key derivation functions is explained in Section 2.7.6.

2.1 Android Architecture

Android is a modern open-source operating system (OS) which enjoys great popularity. It is targeted for mobile devices but can operate on many different platforms, such as cars, TVs, and watches. Originally intended for digital cameras, Google has transformed Android into one of the most popular operating systems for mobile devices and beyond. Since the initial release in 2008, Android has seen numerous updates which have introduced new features and corrected bugs. As

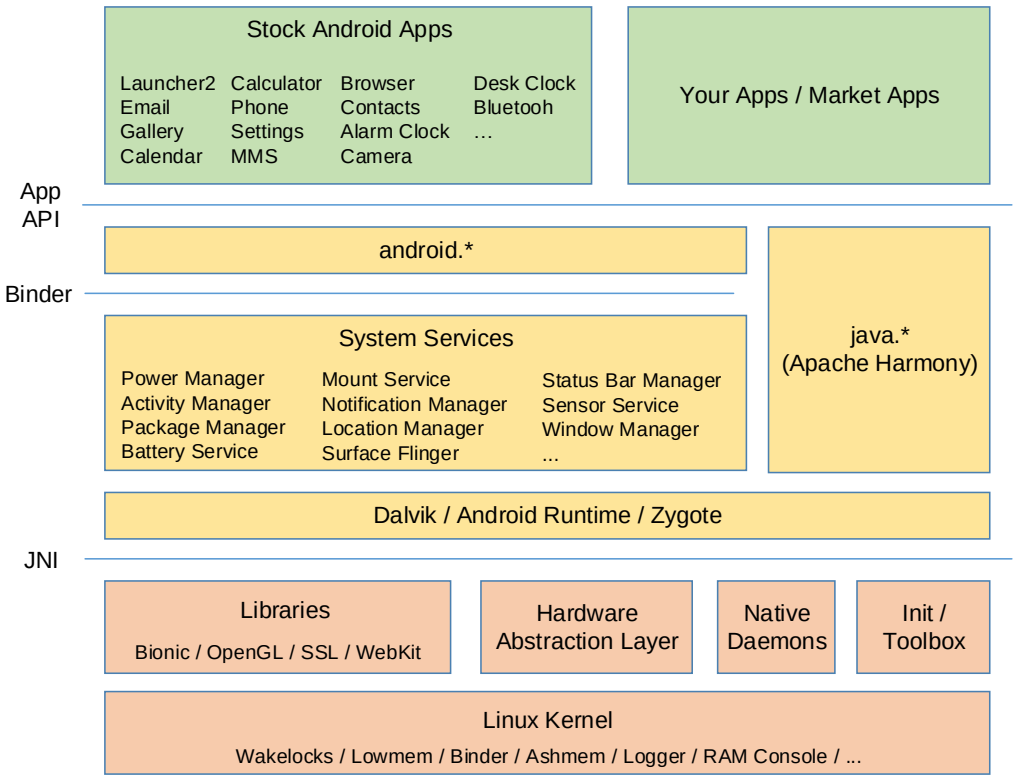

Figure 2.1: Android System Architecture

of December 2014, it held a market share of 76.6%[1] on the global smartphone market. The high adoption rate, ubiquitous presence and open architecture emphasize the need for thoroughly performed security analyses.

As illustrated in Figure 2.1, the software-related architecture of Android can be divided into multiple layers. Each of the layers supplies the higher layers with specified interfaces in order to provide access to their services. For the core functions and for the interaction with hardware components, Android uses a customised version of the Linux kernel. The figure highlights the Android-specific kernel changes, whereas most of them affect memory, power, and process management. Notable exceptions are the Binder and Logger services which are very tailored to the overall concept of Android.

Due to the fact that all applications are run in sandboxed environments, separated from each other, and without shared memory capabilities, they need a way to securely exchange data. For this kind of inter-process communication (IPC), Binder is used. It implements a client-server model, in which the client broadcasts a communication request and awaits a server to respond. Binder also determines the process ID and user ID of a calling process (client) and hands them over to the callee (server) to let it provide access control.

The other component is the Logger which extends the kernel's logging function by adding additional buffers for Android. They are intended to be used only during the development of Android apps and for debugging purposes. Nevertheless, log calls often occur in productive apps

[1] https://www.idc.com/prodserv/smartphone-os-market-share.jsp

which becomes critical if sensitive data is recorded. In contrast to Binder, the Logger implements no access protection which implies that any application on the system is able to read and write information from / to the global protocols. This has effectively been exploitable until the release of Android 4.1 when Google has undertaken changes to prevent misuse. Since then, applications can only access their own log data.

Next to the kernel, already in user-space, reside system libraries and core services. The shared libraries implement low-level functionality and expose an interface to be used by applications. For instance, the OpenSSL library offers various utilities to facilitate encryption. As the libraries are written in native code, they are potentially susceptible to memory corruption.

The middleware components one layer above are responsible for the execution of Android applications. Due to its relevance for our analysis, we discuss the Dalvik Virtual Machine (Dalvik VM) more detailed in Section 2.3. Among the first processes that are started when the device is booted, is Zygote. It then acts as a loader for all subsequent applications that are run in Dalvik VM instances. Once the Android framework and all dependencies are loaded, the process continuously copies itself for each new Dalvik instance that is spawned. The framework provides API abstractions of Android system services in the form of Java packages (`android.*`). The related services are mostly implemented as standalone programs, running as background processes and waiting for events. For instance, The Telephony Manager awaits and handles phone calls. Using the framework API, applications can interact with this service and access telephony-related information, such as the device identifier number (IMEI), the phone number, or the current mobile cell location.

2.2 Android Applications

Application code is usually developed in Java and leverages the APIs, exposed by the Android framework. In case high-performance code is required, it is possible to extend applications with libraries, written in native C/C++. For user interface components and several types of resources, such as strings, and dimensions, the XML format is used. An Android application is stored in a file with the suffix `.apk` (Application Package Format) and consists of multiple pieces which are packaged as a ZIP container. The Android system exclusively accepts files in this format and, by default, requires them to originate from the official Google Play Store. Applications from other sources ("third-party markets") can only be installed if explicitly enabled by the consumer.

Basically it is possible to split applications into two types: pre-installed and consumer-installed apps. The first category comprises applications which are induced by the manufacturer, such as for messaging, the device's camera, or email. These apps are stored in the directory `/system/app/` and typically can not be erased in case they are holding elevated user privileges. The second category, consumer-installed apps, reside in the directory `/data/app/`.

2.2.1 Contained Files

In the following, we briefly denote the content composition of Android application files (`.apk`) and outline their intention:

- **AndroidManifest.xml**

  The Android manifest file contains essential meta-data, needed to execute applications within a protected environment. It lists all required permissions, application components, and defines listener and receiver trigger handlers.

- **assets**

  A directory applications can use to store raw asset files. Files put in this directory are added to archives without being modified. For example, games could archive textures in the `assets` folder and, on demand, load them using the *AssetManager* system service. In contrast to convential resources, assets have no ID assigned.

- **classes.dex**

  The `classes.dex` file comprises all classes compiled to Dalvik bytecode. A more detailed insight on the Dalvik Virtual Machine and Dalvik bytecode is provided in Section 2.3.

- **lib**

  The library directory encloses compiled libraries that base upon native C/C++ code. Due to the fact that native code is platform-specific, individual files need to be built for each processor architecture. The resulting libraries are shared-object files (`.so`) and have to be stored in sub-directories, which are named after the according architecture. Usually separate versions are created for ARM ("armeabi"), ARMv7 and newer ("armeabi-v7a"), x86, and MIPS processors. Although the file size of Android applications is no critical factor, it is significantly increased by this replication of executable code.

- **META-INF**

  The `META-INF` folder contains the following files:

  - `MANIFEST.MF`: Enumerates all files of the application package and the corresponding SHA-1 checksums. It is created when all files are bundled into one `.apk` files.
  - `CERT.SF`: This signature file is added during the code signing step, carried out by the `jarsigner` tool [33]. `CERT.SF` lists the data to be signed and is structurally very similar to `MANIFEST.SF`. It includes the digest of the entire manifest file ("SHA1-Digest-Manifest") and digests of all individual entries in the manifest file. With Java 7, Oracle changed the default algorithm of `jarsigner` to *SHA256withRSA*. However, Android stills recommends[2] to use *SHA1withRSA* because the new default algorithm is not supported before Android 4.2. When invoked, `jarsigner` applies the provided algorithm and signs `CERT.SF`.
  - `CERT.RSA`: Contains the name of the used digest algorithm, a digital signature value of the signed `CERT.SF` file, and the developer's signing certificate in the PKCS#7 format[3]. The suffix `.RSA` indicates that the signature block file was generated using the RSA algorithm. In that case, the public key can be used to verify the signature of `CERT.SF`.

- **res**

  This directory comprises all resources the application is equipped with. Examples include user interface layouts in XML format, animations, and graphics. Equally, presets of colours, menu designs, and strings are stored in the `res` directory.

- **resources.arsc**

  A file containing resource meta data in binary format. All used resources are listed in a table with their resource ID and meta information.

[2] https://developer.android.com/tools/publishing/app-signing.html
[3] http://tools.ietf.org/html/rfc5652

2.2.2 Components

The Android API Guide [4] explains that basically four different types of components are involved when an application is run: activities, services, content providers and broadcast receivers. Each component serves as endpoint in inter-app communication and is characterised by its distinct scope and lifecycle.

Activities

An activity is employed to draw a graphical user interface (GUI). It renders a predefined layout and defines possible events. During interaction with an application, activities handle user inputs and trigger visually displayed reactions. The capabilities of activities in terms of security-critical aspects are regulated by permission settings in the Android Manifest

According to the official guidelines, different tasks should be encapsulated in separate activities. For example, when a consumer needs to login prior to performing other actions, the login procedure should be enclosed by one activity. This concept makes it possible to reuse activities in other sequences and benefit from not needing to write redundant code. Besides, it keeps user inputs mostly self-contained in activities, leading to less data sharing between components and thereby tends to make applications more secure. However, activities can work together, construct a consecutive user experience, and exchange data using *Intents* (see Section 2.2.3). Aside from sharing data with components of the same application, they can be exposed in order to be callable from other applications.

Services

Services have no UI and stay running in the background, even if the appendant application is currently not used. For example, Android natively includes *SMSReceiverService*, a service designed to handle incoming short messages. Also, music playing in the background would be a typical application scenario. The service is once started by an activity and continues running even if the starting component does no longer exist.

Due to the fact that services live independent from the corresponding application, inter-process communication is possible. For obvious reasons, the service needs to be secured in terms of access restrictions to prevent misuse. Therefore, services are declared in the Android Manifest of an application, attached with parameters that define the extent of exposure. Precisely, the service visibility, the required permissions and the process type qualify the security of a service.

Content Providers

A content provider enables the structured storage of data. In most cases, a SQLite database is used to archive application data but basically any persistent storage (file system, cloud, etc.) is suitable. For data access, the content provider abstracts the underlying file or database and provides a high-level interface. The self-contained design also enables other applications to read and modify data. For this to be possible, data sharing needs to be explicitly enabled. Similarly, a content provider can regulate access by prescribing that the client application is granted certain permissions.

A popular example cites a content provider which manages the consumer's contacts. Any application holding the required permissions can access the content provider and read or write

[4] https://developer.android.com/guide/components/fundamentals.html

contact informations.

In case of an underlying SQLite database, reading from the data source implies performing a SQL query. Assuming that user input is added unvalidated to the query, SQL injection is feasible and potentially sensitive data could be leaked to an attacker.

Broadcast Receivers

Broadcast receivers await system-wide *Intents* to be emitted and trigger further actions upon receipt. For example, after the Android device has finished booting, it sends the announcement `ACTION_BOOT_COMPLETED` out to all listening broadcast receivers. Depending on the concrete implementation, further tasks could be performed, such as installing alarms or starting applications. Broadcast receivers are defined in the Android Manifest of an application and mostly act as a gateway to other components.

2.2.3 Intents and Intent Filters

Android uses intents to asynchronously exchange information between activities, services, content providers, and broadcast receivers. Apart from facilitating the communication between components, intents are mainly used to start activities and services, and to emit announcements. Prior to a transmission, data needs to be serialised which enables the broadcast of arbitrary information. Before an Intent is sent out, a receiving component has to exist, capable of handling the incoming event. Due to the fact that any application can address an exposed receiver, it is evident that the targeted component needs to validate the intent data before it is processed further. If no component accepts the intent, the sending application crashes.

```
1  // Create the text message with a string
2  Intent sendIntent = new Intent();
3  sendIntent.setAction(Intent.ACTION_SEND);
4  sendIntent.setType(HTTP.PLAIN_TEXT_TYPE); // "text/plain" MIME type
5  sendIntent.putExtra(Intent.EXTRA_TEXT, "A text message");
6
7  // Verify that the intent will resolve to an activity
8  if (sendIntent.resolveActivity(getPackageManager()) != null) {
9      startActivity(sendIntent);
10 }
```

Listing 2.1: Implicit Intent example [45]

Basically Android distinguishes between *explicit* and *implicit* intents. If the identifier of a receiving component is known, the sender is advised to perform an explicit intent by sending the intent to solely one specific component. For instance, if an activity hands over data to another activity of the same application, it may specify the recipient's class name. In contrast, an implicit intent is identified by its type of action rather than by an identifying name. When the intent is emitted, the Android system tries to determine the best suited receiver component, able to properly execute the specified action. In order to accomplish this goal, implicit intents are tested against so-called *intent filters*.

Listing 2.1 demonstrates the construction of an implicit intent. First, a constant string describing the intended action is set. In the example, `Intent.ACTION_SEND`[5] is designed to send data to

[5] https://developer.android.com/reference/android/content/Intent.html#ACTION_SEND

other applications which will share it via email or social networks. Specifying the MIME type in the next line is no requirement but can help to address the most appropriate receiver component if multiple are available. Finally, the serializable intent content is inserted. The invocation of `resolveActivity()` prevents that the sending application crashes in case no component can handle the intent. Upon calling `startActivity()`, Android looks up the best matching receiver component by examining all available intent filters.

```
1 <intent-filter>
2   <action android:name="android.intent.action.SEND"/>
3   <category android:name="android.intent.category.DEFAULT"/>
4   <data android:mimeType="text/plain"/>
5 </intent-filter>
```

Listing 2.2: Intent Filter example [45]

Intent filters are employed to describe receiving components of implicit intents. For explicit intents the declaration of filters is dispensable because these intents are anyway delivered to the specified component. Usually, intent filters have to be defined in the Android Manifest , except for broadcast receivers which can alternatively be registered dynamically.

As the XML snippet in Listing 2.2 illustrates, an intent filter is characterized by three elements: action, category, and data. By incorporating the value `android.intent.action.SEND` for the intent action and `text/plain` for the type of data, the given example perfectly qualifies as a handler for the intent, constructed in Listing 2.1.

In opposition to other Android components, intent filters can not be secured. Their use implies the exposure of a component so that any application can send intents to it.

2.2.4 Application Signing

In the course of building an application, it needs to be signed by the developer's signature. Therefore, using public-key cryptography, the developer generates a public / private key pair and signs the application with his private key. The public key is distributed with the application and enables the device to verify the authenticity of the provided package. Furthermore, it resembles a protection mechanism to prevent unsolicited or forged application updates. Those are only possible if the update and the already installed basis exhibit the same signature. Assuming a developer offers multiple applications, all need to incorporate the same signature in case they are intended to exchange data among each other. For pre-installed applications, Android uses a special signing key which indicates to the system that the applications might obtain elevated privileges.

In contrast to the traditional employment of a public-key infrastructure (PKI), all Android applications are self-signed by the developer who, at the same time, represents the certificate authority (CA). As a consequence, there is no chain of trust which would end with a root authority. The consumer is constrained to trust the developer's honesty.

After all, application signing on Android must not be confounded with code signing. Due to the fact that Android supports loading additional code at runtime, it seems unfeasible to sign code when deploying Android applications. However, it would allow the adoption of advanced memory protection mechanisms, as pursued on Apple iOS, for example. On the competitor system all applications are signed by a trusted party (e.g. Apple) and only signed code is executable.

2.3 Dalvik Virtual Machine

Although smartphones provide extensive technological features nowadays, Android is targeted to also run efficiently on devices that suffer from considerable hardware constraints. The Dalvik Virtual Machine (DVM) is a managed runtime environment for Android applications which was designed to overcome hardware limitations, such as slow CPU, little RAM, and no swap space in the best way possible. It is specifically designed to be run on ARM processors with 16 registers and also employs a just-in-time (JIT) compiler, arranged for this architecture.

Android Applications are developed in Java but not executed in the traditional stack-based Java Virtual Machine (JVM). Instead, Android uses the register-based DVM to run each application in its own process with a detached memory space. Once the operating system is booted up, the DVM starts and preloads all available libraries. In case an application start is requested, the DVM process, named *Zygote*, replicates itself and executes the demanded application. By not invoking the DVM again, it is possible to share read-only memory pages with the loaded system libraries among all applications, thus reducing the overall memory footprint. Nevertheless, each process is assigned its individual user ID, has only access to a limited set of features on the host system, and can solely modify its own data. Furthermore, Android boosts the performance of Dalvik using a "fine-granularity JIT" which analyses and optimizes the most necessary chunks of bytecode to native code at runtime.

A register-based VM implies that operands are stored on CPU registers, rather than being saved on a stack data structure. For example, when two numbers are added, the result is stored in the third register. As the overhead of pushing and popping values from the stack does not apply, instructions execute faster. However, as it is not possible to rely on a stack pointer, operand locations need to be provided explicitly for all instructions. In fact, Dalvik possesses 2^{16} virtual registers which can serve as locations in the VM's memory. In practice, it appears to be uncommon to use more than 16 registers, though. For the storage of typical bit values, such as integers or floating point numbers, registers are considered 32-bit wide. 64-bit values are put into two adjacent registers.

2.3.1 Dalvik Compilation

Traditionally the Java compiler produces `.class` files from Java source files that can be executed on the stack-based JVM. Dalvik, however, requires the entire bytecode to be contained in only one file which is organized in the proprietary Dalvik executable format.

Before Java code can be executed using the DVM, it needs to be translated to Dalvik bytecode. Figure 2.2 delivers a conceptual overview of the compilation process for DVM bytecode. First, the Java compiler generates JVM bytecode and stores each Java class into its own class file. Subsequently, the Dalvik compiler *dx* consumes the Java classes and recompiles them to Dalvik bytecode. This includes the aggregation of all class files into one output file, named `classes.dex`. During the compilation process the three basic elements of all Java class files are translated, reconstructed, and interpreted. The *constant pool* allocates the constant values, used by a class. The *class definition* comprises basic information, such as class names and access flags. Lastly, the *data* section includes the actual application code and also provides meta information about the used class and instance variables.

By reusing and eliminating repetitive function signatures, code blocks, and string values, the Dalvik compiler manages to effectively reduce the uncompressed bytecode size. As a result, all parts are merged into a single executable file which serves as input for the DVM interpreter.

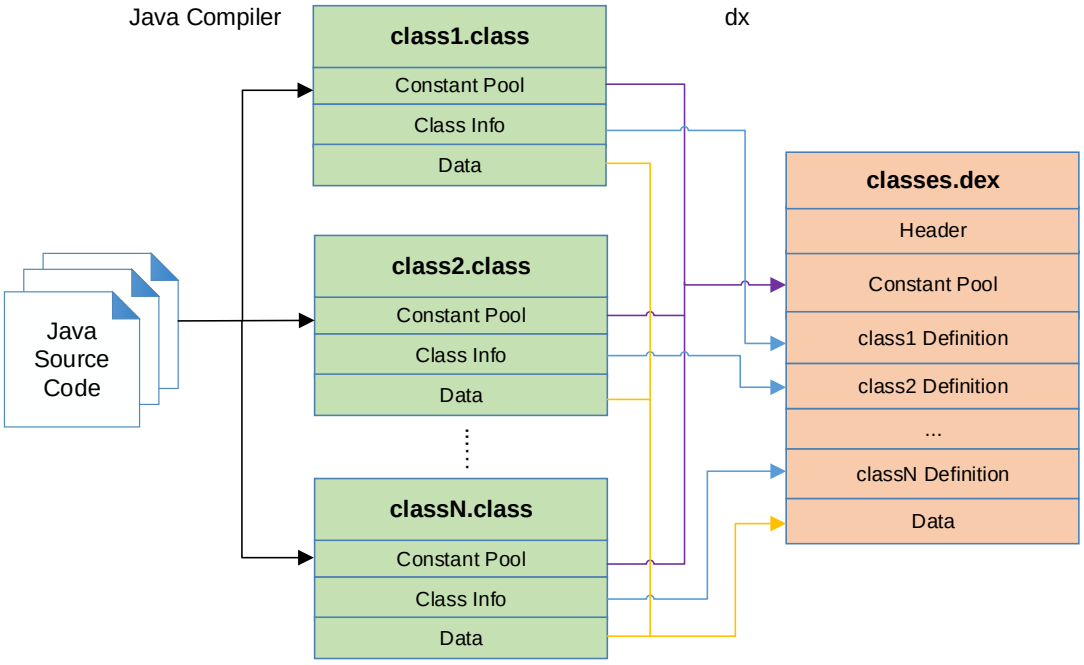

Figure 2.2: Compilation of Android applications

2.3.2 Dalvik Bytecode

Both Dalvik and the JVM comprehend a comparably large set of high-level instructions. Unlike Java bytecode, which is limited to 8-bit stack instructions, Dalvik uses a 16-bit instruction set and implements registers as 4-bit fields. This means that the DVM loads twice as much bytes per read operation than the standard JVM which results in a significant rise of the interpreter speed.

Operating directly on the registers, the Dalvik instruction set consists of 218 opcodes. The variable-length instructions can address either the first 16, 256, or all 2^{16} virtual registers. Assuming an instruction would need to use a register which is not within the available range, the register contents might be moved to a lower register first. As Dalvik instructions often include source and destination registers, they tend to be longer than Java instructions.

In total, Dalvik provides 31 different opcode formats whereas each opcode is identified by a unique, consecutive ID. The Android documentation includes a listing of all opcodes and describes their semantic [6].

2.3.3 Dalvik Executable Format

Android applications include the program code in a file, named `classes.dex`, which is organized in the Dalvik executable format. As depicted in Figure 2.3, the file is split into multiple segments.

The *header* introduces the executable file by specifying the file type using a magic number, and the format version number. Apart from stating the 32-bit CRC checksum of all bytes (except the first 12), it also lists the sizes of the proximate segments and their start addresses. The subsequent *string_ids* section, identifies all strings that are employed either for internal naming or used as

[6] https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

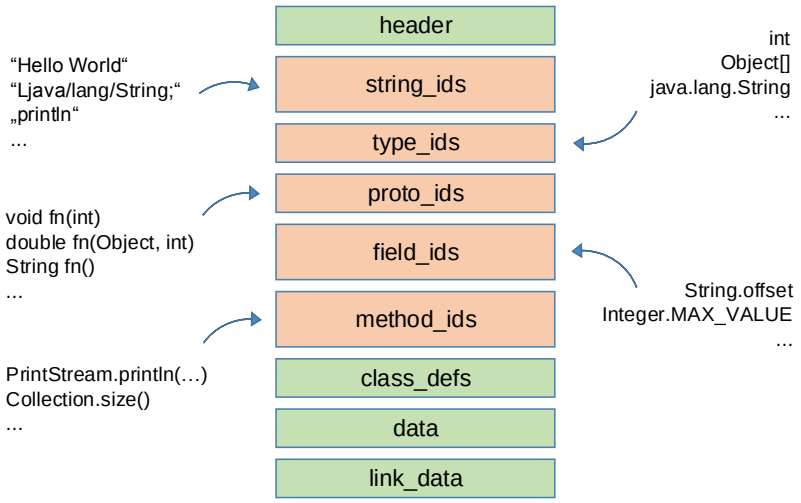

Figure 2.3: File layout of classes.dex

constant string objects. Each string is marked with a unique ID that points to the actual memory location of the string in the *data* segment. Similarly, the *type_ids* block comprises all occurring type identifiers. It enumerates the different type descriptors of classes, arrays, and primitive types, referenced in the current file.

The ensuing sections *proto_ids*, *field_ids*, and *method_ids* list all method signatures, field, and method identifiers. The *proto_ids* block subsumes all prototypes, referenced in the classes. Each entry is composed of three parts: a short descriptor of the prototype, including return and parameter types, a return type ID, and an address offset to determine the associated parameter list in the *data* segment. Likewise, a *field_ids* entry contains the ID of the related class, the ID of the data type, and the ID of the field name. Using the same scheme, *method_ids* entries describe all referenced methods.

All class definitions are listed in the *class_defs* section. The enumeration of referring classes is preceded by the corresponding superclass and a list of all implemented interfaces. As a convention, same-named classes must occur only once in this list. The *link_data* block contains data, used in statically linked files. The exact format of this data is not specified.

The *data* area holds the actual informations, referenced from other tables. For instance, data of the previously mentioned *class_defs* entries, are stored as *class_data_item* objects. Packed in *code_item* objects, this section also models all bytecode instructions, accompanied by meta data about the involved registers and debug information.

During the installation of an Android application, the integrity of the `classes.dex` file is verified using predefined rules. For example, not more padding bytes than required must be used and they have to consist of zero values only. Likewise, strings must not occur twice and are required to be sorted alphabetically. Furthermore, it is ensured that the provided bytecode is syntactically valid and correctly applied. Due to this one-time verification, the Dalvik interpreter can safely disregard many potential error cases during the execution.

Subsequently, applications are optimized for the currently used processor. Thereby, instructions and data are aligned for the underlying system and empty methods are pruned. Moreover, the optimizer performs static linking of method calls and instance fields, and exchanges calls to certain Java library methods (i.e. `java.lang.String.equals()`) with their native representation.

2.4 Smali Language

Smali is a mnemonic language for Dalvik bytecode. It facilitates the analysis of Android applications by representing bytecode as human-readable and parseable source code. Therefore, the tool *baksmali* [19] disassembles the `classes.dex` file and rebuilds the structure of the original Java source code. In contrast to other disassemblers, such as *dexdump* [7] or *dedexer* [8], Smali imitates the original directory structure and outputs each class into its own `.smali` file. The same applies for nested classes which are renamed and stored separately. Internally *baksmali* pursues a recursive traversal approach [42] to find instructions. Thereby, the disassembler continuously follows the control flow of each branch and function call, encountered in the application.

2.4.1 Registers

Similar to Dalvik bytecode, Smali is constrained to use 32-bit registers for any data type. Each method is granted a fresh set of 2^{16} virtual registers whereas registers behave like local variables in a way that they do not affect invoking methods.

While Dalvik addresses all registers with the prefix `v`, Smali introduces the letter `p` to prefix registers that contain method parameters. Due to this convention, ordinary registers are easily distinguishable from parameter registers. For instance, in *static* methods the register name `p0` describes the first parameter, passed to the method. The second would be denominated `p1`. However, in the case of *non-static* methods, `p0` always holds the instance the method is being invoked on (`this` object). Passed method parameters are located in registers beginning with `p1`.

64-bit values are put into two adjacent registers. Assuming that such a value is used as a method parameter, both containing registers need to be provided to the invoking instruction.

2.4.2 Instruction Syntax

The Smali syntax is inspired by Jasmin [9] and incorporates the entire Dalvik instruction set. For most instructions, the first argument is the destination register, followed by the source register. As depicted in Listing 2.3, immediate values, such as constant strings and integers, are directly embedded into the code. Thereby, Smali deviates from the Dalvik syntax which references strings by their ID, instead of inlining them.

```
1 const-string v0, "Hello World!"
2 add-int/lit16 v1, v6, 11138
```

Listing 2.3: Smali syntax

In the given example, the first line assigns the string value "Hello World!" to the register v0. The second line adds the 16-bit signed integer constant 11138 to the register v6 and stores the result in register v1.

Dalvik incorporates several instructions that can operate on multiple data types. In order to distinguish between the different sizes, opcodes are suffixed with their type. For example, the instruction `aget-boolean` is used to load a boolean value from an array. The instruction `aget-wide` would perform the same operation for 64-bit values.

[7] https://developer.android.com/sdk/
[8] http://dedexer.sourceforge.net
[9] http://jasmin.sourceforge.net

2.4.3 File Structure

As illustrated in Listing 2.4, Smali files replicate the structure of Java classes. The first line contains the fully-qualified class name. Based on that, the related directory structure is deduced in which the Smali file resides on the system. The subsequent header lines indicate class inheritance.

Next, class members are declared. The employed modifiers define if the fields act as instance variables or statically defined members. The `.method` keyword is used to start method declarations. Apart from the method name, the parameter types, the return type, and the access modifiers specify the method signature, required for the virtual machine to target the correct method. The keyword `.locals` declares the amount of variable (registers), needed for this method. One or multiple `.param` lines specify the registers which hold the method parameters.

```
1  .class modifiers... Lsome/package/SomeClass;
2  .super Lsome/package/SuperClass;
3  .implements Lsome/package/SomeInterface;
4
5  .field modifiers... fieldName:Lfield/type; = "finalFieldValue"
6  ...
7
8  .method modifiers... methodName(Larg/types;)Lreturn/type;
9      .locals X
10     .param pX
11     ...
12
13     instruction...
14     instruction...
15     ...
16 .end method
17 ...
```

Listing 2.4: Smali code example

As can be seen in the sample code, the letter L prepends all class names. This also applies to non-primitive data types, such as objects and arrays. In contrast, primitives are represented by single letters. Table 2.1 illustrates the available Dalvik data types and their counterpart in Java.

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

Table 2.1: Dalvik data types and their Java equivalents

2.5 Reverse Engineering of Android Applications

Using Reverse Engineering it is possible to learn implementation details about applications. Essentially, the goal is to transform an existing representation of a program into an easier understandable form. In the case of Android applications the optimal result of this process would be the complete reconstruction of the original Java source code from Dalvik bytecode. Detailed background information on the development of a custom decompiler is provided by Nolan [30].

In a first step, the decompilation process of Android applications usually consists in converting the Dalvik executable to Java's `.class` format. Due to the fact that this operation yields no source code in a high-level language, it can be seen as a code transcription, rather than a decompilation. Popular tools performing this kind of application retargeting include *dex2jar* [10], *Dare* [32], and its predecessor *ded* [31]. Using predefined rules and constraint solving mechanisms, the mentioned programs translate bytecode of the register-based Dalvik VM to bytecode for the stack-based JVM. The resulting `.class` files are typically packaged as a `.jar` container. Finally, various tools, such as *JD-GUI* [11], *JAD* [12], or *Dava* [13], accomplish the actual decompilation to Java source code. Apart from decompiling applications via an intermediate step, Dalvik bytecode may be directly translated to source code. For example, this is achieved by the commercial software *JEB* [14].

During the compilation of Android applications, XML resource files and the Android Manifest are transformed into binary files. Thereby, string values are extracted from the XML files and collected in the file `resources.arsc`. Programs like *AXMLPrinter2* [15] or *axml* [16] reverse this process and output readable XML files.

apktool [17] performs a combination of resource decoding and Dalvik bytecode disassembling. *AXMLPrinter2* restores XML files and the disassembly using *baksmali* (see Section 2.4) yields Smali files. As application analysis benefits from the strong affinity of Smali code with the original Dalvik bytecode, we employ *apktool* for this thesis.

2.5.1 Prevention

Presumably to protect intellectual property, developers often apply code protection mechanisms in order to impede analysis. For Android, several code obfuscation techniques are established which modify the Java or Dalvik bytecode. As a consequence, decompilers are tricked and may refuse to work. The most widespread obfuscator for Java bytecode is *ProGuard* [18]. It optimizes the code, scrambles the names of identifiers and reduces the code size by removing unused classes as well as dead code.

Operating directly on Dalvik executable, the tool *dalvik-obfuscator* [19] inserts junk bytes into the bytecode. Focussing on the encryption of strings, application resources, and libraries, the commercial obfuscator DexGuard [20] applies various methods on both source and bytecode level to hinder decompilation.

[10] https://code.google.com/p/dex2jar/
[11] http://jd.benow.ca
[12] http://en.wikipedia.org/wiki/JAD_(JAva_Decompiler)
[13] http://www.sable.mcgill.ca/dava/
[14] http://www.android-decompiler.com
[15] https://code.google.com/p/android4me/
[16] https://code.google.com/p/axml/
[17] https://code.google.com/p/android-apktool/
[18] http://proguard.sourceforge.net
[19] https://github.com/thuxnder/dalvik-obfuscator
[20] http://www.saikoa.com/dexguard

2.6 Program Slicing

For a better understanding of complex computer programs and easier debugging, Weiser [48] introduced the concept of program slicing. The basic idea is to reduce programs to only those code statements which are essential for a particular computation. This implies that slicing requires the actual program code or an adequate representation thereof, suited for the decomposition into smaller pieces.

2.6.1 Terminology

A *program slice* consists of all statements that potentially affect the variables at a certain program point. It forms an independent and minimal program which models the original program's behaviour. However, Weiser has proven that it seems unfeasible to determine minimal slices. As a consequence, a more practical approach has been elaborated, based on analyzing the *data* and *control flow* of applications. Each slice relates to a *slicing criterion* $C = (i, V)$ which defines the values of interest. The pair (i, V) refers to a specific statement in the code and the set of variables in demand. Starting with the slicing criterion, all code lines are added to the slice that influence the statements which are already in the slice. The procedure is repeated until all necessary statements are added. Since the number of code lines is finite, slicing always terminates.

Slices can be obtained using *static* or *dynamic* analysis. They are fundamentally different as the former makes no assumptions about the program's input, whereas the latter is contingent on a particular test case. Section 2.6.3 illustrates the differences between the two types. Furthermore, slicing can be conducted in *backward* or *forward* direction. When slicing backward, all statements that have led to the slicing criterion are determined. In contrast, forward slices include statements which are directly or indirectly derived from the slicing criterion. Therefore, each direction requires a distinct slicing algorithm. Section 2.6.4 elaborates the differences in detail.

2.6.2 Precision and Scalability

Since most programming languages differ in semantics and features, slicing algorithms are individually adapted for the designated use. Tip [46] highlighted "procedures, unstructured control flow, composite data types and pointers, and concurrency" as particularities that need to be taken into account when applying a slicing algorithm. Further challenging tasks to cope with include slicing precision and algorithm scalability.

Slicing precision is limited by the fact that it can not be determined whether there is a program input that triggers the execution of a statically detected path (execution trace). Similarly, slices should not contain paths which are never reached during the program execution. These unnecessary statements, added by non-realizable paths, tend to reduce the expressiveness of a slice. To remedy this problem, Ottenstein and Ottenstein [34] have proposed *program dependency graphs* (PDG). Thereby, the nodes of a directed graph represent code statements and the edges control or data flow dependencies. PDGs enable a precise computation of slices by testing the graph reachability. Although efficient interprocedular slicing is possible with PDGs, the costs for the previously necessary graph construction are not negligible and increase with the size of the program.

The scalability of a slicing algorithm is given if the time needed to compute a slice depends roughly on the size of the slice, rather than the size of the program. If only a few slices are computed or the results can be expected to be small, it seems reasonable to follow an *on-the-fly* approach where information is calculated at the time it is needed.

2.6.3 Slicing Variants

Weiser originally proposed *static slicing* as a method to compute slices without making assumptions about the program input. Korel and Laski [26] presented an approach for *dynamic slicing* which calculates slices with respect to a fixed input.

Static Slicing

Static slicing can be used to determine all code statements of a program that *may* affect the slicing criterion. Static slices are valid for all input values and do not evaluate predicates to true or false during the analysis. In other words, only statically present information is processed. Basically they cover all possible execution paths and allow conclusions to be drawn about the functionality of the program. By making conservative assumptions, comparably big slices are constructed.

The surveys of Tip [46] and Xu et al. [50] sketch three different concepts for static slicing:

- **Data flow equations**
  The concept of Weiser is based on control flow graphs (CFG). Slices are determined by investigating the data flow between all relevant variables. Therefrom, relevant statements are derived which are part of the slice.

- **Information-flow relations**
  By considering slicing as an information-flow problem, Bergeretti and Carré [7] proposed another approach. The information-flow relations are derived by evaluating formulas on the source code. For each type of code statement, an individual formula evaluates the statement's influence on a variable. The result is a $(0, 1)$ integer matrix, indicating the relations between code lines. Based on this data, a backward or forward slice can be deduced.

- **Dependency graph based**
  The third approach defines slicing as a graph reachability problem. The algorithm of Ottenstein and Ottenstein [34] constructs a program dependency graph (PDG) whose nodes represent control predicates and assignment statements, such as read or write operations. Derived from the entire program, the graph takes only realizable paths into account and creates slices by solving the graph reachability. The slicing criterion is represented as a graph node in the PDG. A slice involves all PDG nodes from which the node with the slicing criterion can be reached.

All listed approaches are capable of slicing simple programs, which consist of only one function, a well-formed control flow and work without concurrency. However, they are not equally qualified to perform slicing on sophisticated programs. For example, Android (Java) applications may contain the aforementioned unstructured control flow (using exceptions, `break`, `continue`, and `goto` statements), multiple methods, complex data types and concurrency.

Data flow equations and dependency graphs consider unstructured control flow and multiple methods appropriately. While data flow equations fail at handling concurrency, multiple threads can be represented using dependency graphs [51]. Information-flow relations are unable to handle unstructured control flow and concurrency. Slicing over multiple methods is achievable by any of the presented approaches.

Dynamic Slicing

Dynamic flow concepts derive slices by executing the program with a certain input with respect to a slicing criterion. The resulting execution history forms a program whose behaviour does not differ from the original program. However, as the execution trace is specific to the induced test case, dynamic slices can be significantly smaller than the corresponding static slice.

A dynamic slicing criterion consists of three parameters $C = (x, I^q, V)$. Variable x declares the particular test input. I^q denominates the statement in the execution trace from which to start slicing. Usually, I refers to the line of code in the original program and q enumerates the index in the execution history. The variable V contains the set of variables in demand.

In order to compute a dynamic slice, an execution trace needs to be created by running the program with the test parameters. Subsequently, a directed graph is drawn with the executed statements as nodes. The edges represent data and control dependencies. Starting with the slicing criterion, all dependencies are traced back and visited nodes are added to the slice.

Comparison

Table 2.2 highlights the differences between static and dynamic slicing by applying both techniques on a sample code. Depending on the value of the parameter `op`, the given program calculates the sinus or cosinus of the predefined floating-point numbers `x1` or `x2` and stores the result in the variable `y`. In both cases, the resulting slice with respect to the slicing criterion is printed in bold font.

The static backward slice is performed for the slicing criterion $C = (11, \{y\})$. Initially the code statement at line 11 is added to the slice. Subsequently, all preceding code statements, which influence the statements that are already contained, are appended. Due to the fact that all code lines have an impact on the value of `y`, the static slice comprises the entire program.

In contrast, dynamic slicing requires a particular test case, which is given by *op = sin*. By running the program with the supplied test case, an execution history is created, enclosing solely those code lines that are executed with the particular input. The dynamic slice then comprises all statements which actually affect variable `y` in line 11 under the premise of the input *op = sin*. As the else-clause in line 8 is never matched, the statements in lines 3, 8 and 9 are not part of the slice.

	Static backward slice for *(11, {y})*	**Dynamic slice for the input *op = sin***
1	`void mathFunction(String op) {`	`void mathFunction(String op) {`
2	`  double x1 = 5.4;`	`  double x1 = 5.4;`
3	`  double x2 = 10.2;`	`  double x2 = 10.2;`
4	`  double y = 0;`	`  double y = 0;`
5		
6	`  if (op.equals("sin"))`	`  if (op.equals("sin"))`
7	`    y = Math.sin(x1);`	`    y = Math.sin(x1);`
8	`  else`	`  else`
9	`    y = Math.cos(x2) + 0.8;`	`    y = Math.cos(x2) + 0.8;`
10		
11	`  System.out.println(y);`	`  System.out.println(y);`
12	`}`	`}`

Table 2.2: Static and dynamic slicing in comparison

2.6.4 Forward and Backward Slicing

The slicing concepts, presented in the previous section, compute slices by traversing the program back to the start. Thereby, they are looking for all preceding code statements that have influenced a variable at the slicing criterion. This retrospection is commonly known as *backward slicing*.

In contrast, *forward slicing* intends to find all parts which are influenced by the variable at the slicing criterion. This implies that a forward slice contains all statements which depend on a value computed at the slicing criterion. In terms of dynamic slicing, the initial value of the slicing criterion may be used to determine if subsequent statements are executed or not.

Basically, backward and forward slices can be computed similarly. A practical approach for interprocedural forward slicing using dependency graphs has been shown by Horwitz, Reps, and Binkley [22]. They have demonstrated a multi-stage procedure to traverse the edges of the graph from source (slicing criterion) to target.

2.6.5 Data Dependencies

Program slicing intends to determine data dependencies by analysing the data flow of variables. A data dependence between two code statements (or graph nodes, respectively) is given if an instruction is dependent on the data of a preceding statement. Aho et al. [3] illustrate three requirements that have to be satisfied for a data dependency to exist:

1. A statement D defines a variable x.
2. Another statement U uses x.
3. The control flow reaches U after D without encountering an intervening definition of x.

In case the statement U can be reached via multiple execution paths, its value depends on all definitions of D that have a clear path to U. Assuming that the order of the statements would be altered or reversed, the data flow would change as well and a dependency might be nullified.

Definition-Use Chains

Definition-use (or short def-use) chains express data flow relationships. They model read and write operations on variables by determining definitions and uses of variables within the data flow.

A def-use chain starts with a variable definition. This write operation is unambiguous since the variable is assigned with a new value. Further definitions within the data flow may either replace the variable content with an entirely different value, or alter the existing value. In the first case, a previous definition would be "killed" and subsequent read operations would refer to the new value. The impact of the latter case, however, depends on the underlying data structure. For example, if an element is modified within a list, it is generally hard to decide how the list has changed at all. Although the variable has still the same name, the reference may have changed. This is commonly known as *aliasing* and should to be taken into account when working with def-use chains.

Read access to previously defined variables are denoted as uses. In a particular statement, basically all variables which are not defined, belong to the set of used variables.

A comprehensive approach for interprocedural data flow analysis using def-use chains has been proposed by Harrold and Soffa [20]. Apart from dealing with reference parameters and global variables, they also present concepts to handle aliasing and recursion.

2.7 Cryptography on Android

Android applications use cryptography to protect sensitive data, such as personally identifiable information (PII), from being leaked to unrelated parties. Therefore, the platform provides extensible cryptographic APIs which can be used to perform encryption and decryption of data.

Since the goal of this thesis is the analysis of cryptography-related functions, we explain the basic principles of cryptography and relate to their utilisation in Java-based applications.

2.7.1 Architecture

By including the Java Cryptographic Architecture (JCA), Android supports a well-established set of security APIs. The JCA is provider-based which means that the interfaces may be implemented by multiple cryptographic engines in the background. For example, if a particular algorithm is requested by an application, the JCA consults all registered providers and returns a new object instance, in case a provider contains a matching implementation.

As depicted in Figure 2.4, Android includes several Cryptographic Service Providers (CSP) to implement cryptographic functionality. The actual providers and available algorithms depend on the used Android version. With Android 4.3 the default provider for common crypto-related operations has been changed to OpenSSL [21]. Prior to that, a pruned version of the BouncyCastle [22] library served as the main provider. Presumably to save space, many algorithms and cryptographic schemes are omitted from the BouncyCastle libraries shipped with Android. For example, with the current Android 4.4 it is unfeasible to use ECDH (Elliptic curve Diffie-Hellman) for the secure key exchange between two parties. Similarly, the authenticated encryption scheme AES-GCM is not available on Android, although it is implemented in both OpenSSL and BouncyCastle.

A remedy for the mentioned shortcomings is provided by the SpongyCastle [23] library. It contains the full version of BouncyCastle and may be packaged with any Android application.

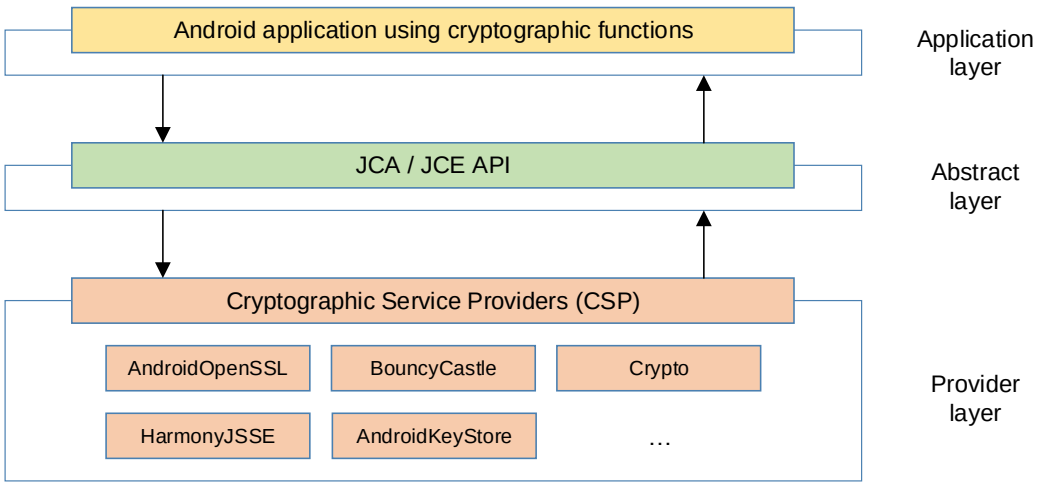

Figure 2.4: Hierarchy of cryptographic operations on Android

[21] https://www.openssl.org
[22] https://www.bouncycastle.org
[23] https://rtyley.github.io/spongycastle/

2.7.2 Symmetric Cryptography

Symmetric cryptography uses the same key for encryption and decryption. In order to keep encrypted data secret, the key must never be leaked to unrelated parties. Among the most frequently used symmetric ciphers are AES, Blowfish, DES, and Triple DES. Typically, symmetric algorithms are split into the categories *stream ciphers* and *block ciphers*.

Stream ciphers

Stream ciphers process a message in small pieces and combine each with a pseudorandom keystream. Based on a predefined key, a bitstream is generated and XORed with each bit of a message. The output is a ciphertext which can be decrypted by applying the same XOR operation another time. In general, stream ciphers are easy to implement and perform faster than block ciphers.

To ensure the security of stream ciphers, it is essential that two messages are never encrypted with the same keystream. As a consequence, a different key or nonce has to be chosen for each invocation of the cipher.

With the current Android 4.4 the only supported stream cipher is RC4. It is implemented by the AndroidOpenSSL and BouncyCastle providers and is mainly used in network applications.

Block ciphers

Block ciphers apply a cryptographic transformation on sequences of bits (blocks), whereas each has a constant size. By iteratively performing substitutions and permutations on each block of a plaintext, a ciphertext is assembled. This process is influenced by an encryption key which is also necessary in order to decrypt the ciphertext. Decryption is usually achieved by reversing the steps of the encryption process. One of the most widely adopted block ciphers is AES. Standardized in 2001 as a replacement for DES, it operates on 128-bit data blocks and supports keys with a length of 128, 192, or 256 bits.

A *mode of operation* defines how multiple blocks of plaintext can securely be transformed to a coherent ciphertext. Beyond the most common modes are ECB, CBC, CFB, OFB, and CTR. Except for ECB, all mentioned modes require an *initialization vector (IV)* which should ensure that the encryption operation emits different ciphertexts, in case a message is encrypted multiple times independently with the same key. Although an IV is usually not a secret, it is indispensable to make an IV unpredictable.

Figure 2.5 illustrates the workflow of encryption in *Electronic Codebook (ECB)* mode. All blocks are encrypted individually and independent from each other. As a consequence, identical message blocks result in identical ciphertext blocks, assuming that the same key is used. Thus, data patterns are not well hidden and message confidentiality may be compromised.

Another mode, *Cipher Block Chaining (CBC)* mode overcomes the issues of ECB mode by XOR-combining each message block with the preceding block's ciphertext. As depicted in Figure 2.6, the encryption of the first message block is dependent on an IV due to the fact that there is no previous ciphertext block yet. Randomly chosen, an IV makes each ciphertext unique. Accordingly, the change of any bit in an IV or message block has an impact on all subsequent ciphertext blocks. The decryption process works in reverse order and needs to be initialized with the same IV, chosen for encryption. However, an incorrect IV only causes the first block of plaintext to be corrupt, while the rest can be decrypted correctly.

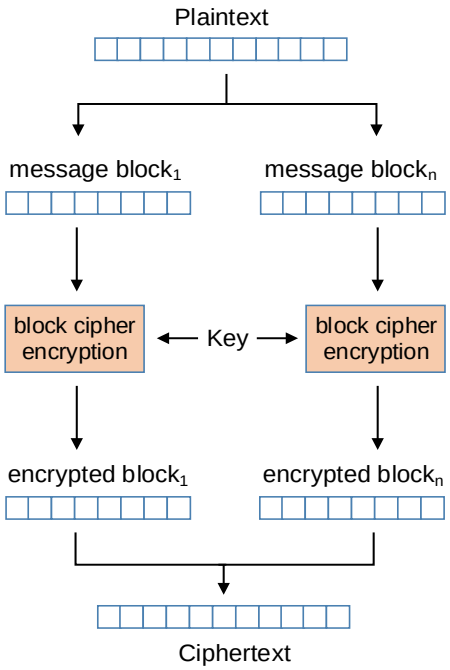

Figure 2.5: ECB mode

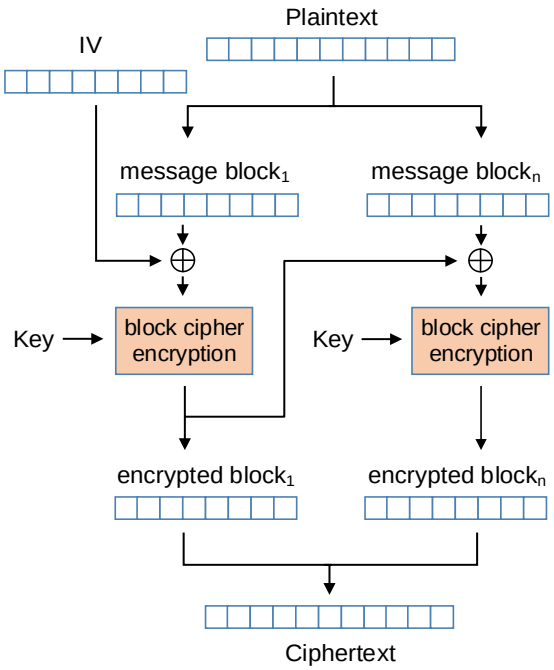

Figure 2.6: CBC mode

The *Cipher Feedback (CFB)* and *Output Feedback (OFB)* modes enable a block cipher to be used as stream cipher. First, a block cipher encrypts the IV with a given key. The encryption outcome is then XOR-combined with a message block and yields a block of ciphertext. While CFB mode uses the emitted ciphertext block as input for the encryption of the subsequent message block, OFB mode takes the encrypted IV. Due to the fact that message blocks are never directly processed by a block cipher, they do not have to correspond to a multiple of the block size. For more detailed information on block cipher modes of operation, we refer to Menezes, van Oorschot, and Vanstone [28].

As the name implies, block ciphers process only units of a fixed size. However, most messages can not be split exactly into blocks with the required length. In order to compensate for the missing data, *padding bytes* have to be added before encryption to close the gap. After decryption, the padding bytes are deleted and the original plaintext length is restored. Basically, any data could be used for padding as long as the length of the plaintext can be recovered. Therefore, several padding schemes exist which specify the values to pad with.

A commonly used padding scheme is PKCS#7 [24]. It works by filling the remainder of a block with bytes, whereas the value of each byte indicates the total number of appended padding bytes. For example, if 4 padding bytes have to be added to complete a block, each of these bytes has the value 4. As such, PKCS#7 can be applied on block sizes up to 256 bytes. A variant of this scheme, limited to a block size of 8 bytes, is known as PKCS#5.

Referring to the source code of Android 4.4, the natively included providers enable encryption of plaintext using AES, Blowfish, DES, and Triple DES. The symmetric block ciphers RC2 and Twofish are implemented as well but can only be used in combination with a key derivation function (see Section 2.7.6).

[24] https://www.ietf.org/rfc/rfc2315.txt

2.7.3 Asymmetric Cryptography

Asymmetric cryptography uses a pair of distinct keys for encryption and decryption. Since the key to encrypt a message can be made publicly available, the concept is also known as public-key cryptography. The decryption key, however, is private and has to be kept hidden. Using the public key, anyone can encrypt data for the holder of the private key without having to share a secret with the other party. Moreover, the private key can be used to create digital signatures over data which are then verifiable using the public key. Although the keys are mathematically related to each other, usually it is unfeasible to deduce the private key from the public key.

One of the most popular public-key algorithms for encrypting messages is RSA. In a first step, each entity has to generate a keypair, consisting of a public and a private key. Therefore, two large and distinct random primes p and q have to be chosen. The product of these numbers $n = pq$ defines the modulus for both keys. Now using the public key e, a message m is encrypted to the ciphertext c by computing $c \equiv m^e \pmod{n}$. The receiver of the encrypted message can decipher the text using his private key d by computing $m \equiv c^d \pmod{n}$.

Compared to encryption with symmetric block ciphers, encryption with RSA involves high computational costs. As a remedy, *hybrid encryption* unifies the convenience of asymmetric ciphers with the performance of symmetric encryption. First, a randomly generated key is used to encrypt a message with a symmetric cipher. Then, the symmetric key is encrypted using an asymmetric scheme, such as RSA. On mobile devices, in particular, performance reasons suggest the use of hybrid encryption instead of plain asymmetric methods.

For further information on asymmetric cryptography, we refer to Menezes, van Oorschot, and Vanstone [28].

2.7.4 Hash Functions

Cryptographic hash functions process data with arbitrary length and compute a fixed-size bit string, named digest or hash value. Typically, they are designed as one-way functions, having the property that a digest can easily be calculated but impossibly inverted. Applications of hash algorithms include password verification, message authentication, and data integrity.

In order to withstand the most common attacks, cryptographic hash functions should have at least the following attributes [38]:

- **Preimage resistance:**
  Given an arbitrary hash value $h(m)$, it must be computationally impossible to find the belonging input message m. For this assumption to be fulfilled, a hash function needs to have a one-way property and the output space of $h(m)$ has to be sufficiently large.

- **Second preimage resistance:**
  Given an input message m, it has to be infeasible to find a second message m', such that $m \neq m'$ and $h(m) = h(m')$.

- **Collision resistance:**
  It has to be impracticable to find two different inputs m and m' which produce the same hash value $h(m)$. To prevent collisions, hash algorithms should distribute digests evenly within a large output space.

Android applications use hash functions mainly for the secure storage of passwords. Archiving passwords in plaintext can result in a severe security breach if the system is compromised. To prevent this circumstance, irreversible hash values of passwords are stored. Therefore, the AndroidOpenSSL and BouncyCastle providers both include various hash functions, which are listed in Table 2.3. MD5 (Message Digest Algorithm) and SHA (Secure Hash Algorithm) are two families of hash algorithms which differ in strength and collision probability.

Although widely deployed, the usage of MD5 is not advisable due to a preimage flaw and multiple collision vulnerabilities. Xie, Liu, and Feng [49] have demonstrated that regular computers are able to find a MD5 collision in less than a second. Similar cryptographic weaknesses were also discovered in SHA-1. Several attacks have revealed that collisions could be computed with a complexity of 2^{61} operations [43]. As a precautionary measure, SHA-1 should not be used in security-critical environments, although no actual collision has been found yet. Introduced with Android 2.3, algorithms of the SHA-2 family (SHA-256, SHA-384, SHA-512) provide at least 128 security bits and are considered to be secure.

Hash algorithm	**Block length (in bits)**	**Output length (in bits)**
MD5	512	128
SHA-1	512	160
SHA-256	512	256
SHA-384	1024	384
SHA-512	1024	512

Table 2.3: Hash functions on Android

2.7.5 Java-Based Random Number Generation

The security of cryptographic applications strongly depends on the generation of unpredictable random numbers. Used as seed for cryptographic functions, salt in hash functions, and key in Message Authentication Codes (MAC), random numbers have to be selected carefully in order to keep risk exposure at a low level. By nature, computer systems are unable to generate truly random numbers and, thus, rely on pseudo-random numbers.

For cryptographic purposes, the JCA on Android provides a Pseudo Random Number Generator (PRNG) which is able to generate a sequence of random bits, according to a predefined algorithm. Based around the SHA-1 message digest, the class `java.security.SecureRandom` implements a strong PRNG, producing non-deterministic output. Internally, it works by digesting a seed value, saving the outcome as internal state and returning the result as pseudo-random data. Unless the seed is explicitly specified using the appropriate constructor, SecureRandom takes the output of the system PRNG `/dev/urandom` for initialization. This source of entropy is unpredictable and suitable for secure use.

Starting with Android 4.2, the default PRNG provider BouncyCastle has been replaced by AndroidOpenSSL. Prior to that, it was possible to set a custom seed, having the effect that generated values were deterministic. With the transition to AndroidOpenSSL, user-given seed values were appended and did no longer substitute the entire internal state of the PRNG. However, in August 2013, Kim, Han, and Lee [25] demonstrated a serious flaw in the PRNG itself. Due to improper

initialization, the system-generated seed suffered from low entropy. As a consequence, random numbers were predictable, even if an application supplied a custom seed.

An alternative PRNG is available with `java.util.Random`. Based on a Linear Congruential Generator (LCG), it produces a predictable sequence of numbers. This is achieved by iteratively solving the recurrence equation $X_{n+1} = (aX_n + c) \mod m$, whereas m, a, c denote constant parameter values and X_0 the initial seed value. Unless manually defined, the system clock is used as seed. Hence, the generated pseudo-random numbers are not cryptographically strong and must not be used for security-critical purposes.

2.7.6 Password-Based Key Derivation Functions

Ultimately, the security of a cryptosystem often depends on secret values or passwords. Since user-entered passwords typically consist of text strings of arbitrary size with variable strength, it is evident that possible shortcomings of passwords need to be taken into account. Disregarding the quality of passwords, applications are urged to store them in a secure way. As already mentioned in Section 2.7.4, a one-way hash function can be used to compute the digest of a secret value. However, hash functions are designed to perform fast and do not necessarily involve a salt. Modern CPU and GPU engines support attackers in constructing a table with all possible passwords and in letting them determine the corresponding password in minimum time.

Key derivation functions (KDF) address these issues and provide a wrapper to derive cryptographically secure keys while slowing down brute-force and dictionary attacks dramatically. Apart from requiring a salt value, KDFs strengthen keys by iteratively applying the derivation technique multiple times. The derived keys are suited to be stored on Android devices and can also be used in cryptographic applications.

Among the most frequently used KDFs is PBKDF2 [23]. Using a pseudo-random function, such as a hash function, a password is transformed to a key of a certain length. As illustrated in Figure 2.7, a password, a cryptographic salt, and the intended length of the derived key are taken as input. The KDF itself is defined by a pseudo-random function and a constant iteration count.

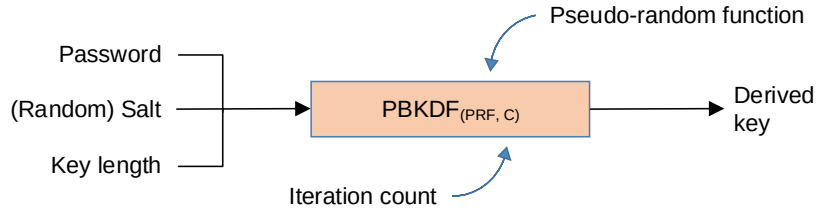

Figure 2.7: PBKDF workflow

The purpose of the salt is to generate unique keys for a given password. Although it is no secret and does not have to be kept hidden, salt values should be generated using a secure PRNG (see Section 2.7.5). In addition, it is possible to derive multiple, different keys from the same password by simply alternating the salt. In either case, the salt should have at least 128 bits [47].

The iteration count is intended to raise the cost of performing attacks on a password. Basically, it should be set as high as possible while keeping performance at a reasonable level. In 2010, a minimum of 1000 iterations has been recommended [47]. Considering the emerging capabilities of mobile devices, higher values might be suitable without noticeable delay.

In Java-based applications, the algorithm for deriving a key from a password is typically identified as `PBKDF2WithHmacSHA1`. The suffix `HmacSHA1` indicates the designated pseudo-random function (PRF). HMAC is a mechanism for message authentication, involving a hash function [25]. Generally, it may be used to verify the authentication and integrity of a message. On Android, HMAC with SHA-1 is the only available PBKDF2 key generator, implemented by the Bouncy-Castle provider. Basically, it is possible to employ HMAC in combination with any recognized hash function as PRF, in order to derive a key, based on a given password and salt.

The most notable alternatives to PBKDF2 include bcrypt [37] and scrypt [35]. bcrypt was specifically designed to be slow while expanding and reusing elements of the symmetric block cipher Blowfish in ECB mode. The iteration count has to be a power of two and is passed as input parameter. Each derived key has a constant length of 192 bits and contains the salt as well as the iteration count. The newer scrypt, in contrast, was invented with the objective of defeating parallel attacks by demanding a significant amount of RAM. It builds over PBKDF2 and the stream cipher Salsa20/8 [26]. For this thesis, bcrypt and scrypt have been disregarded since they are not available on Android.

[25] https://www.ietf.org/rfc/rfc2104.txt
[26] http://cr.yp.to/snuffle/812.pdf

Chapter 3

Related Work

The analysis of security aspects on Android-based devices has attracted a lot of attention in the past years. In this chapter, we discuss related research and highlight aspects, which have influenced this thesis. First, we focus on similar approaches for analysis of Android applications and denote the applied concepts in Section 3.1. Since most of the related Android analysis frameworks do not involve cryptography, related studies are listed separately in Section 3.2.

3.1 Android Application Analysis

When performing static analysis, a predefined set of inspection procedures is applied to an application. Thereby, although the program is never executed and regardless of the actual behaviour, it is possible to draw conclusions based on selected features. In case of Android application analysis, usually either the Dalvik bytecode (see Section 2.3) or a higher-level abstraction, such as Smali code (see Section 2.4) is investigated. Depending on the objective, other available files, such as an application's Android manifest, resources or included libraries, can also be used to influence the analysis. Dynamic analysis, on the other hand, runs Android applications in a virtualised environment and is solely dependent on the application's behaviour.

Hoffmann et al. [21] have presented a framework, named SAAF, to perform static slicing of Smali code. The tool is intended to support manual analysis of applications and implements backward slicing (only). Our CryptoSlice framework has essentially been inspired by SAAF and shares a similar framework structure and backward slicing technique. However, SAAF was unable to meet our requirements in terms of accuracy and implemented features.

One of the most popular tools for reverse engineering and malware analysis is Androguard [1]. It allows disassembling, decompilation and manipulation of Android applications. Besides, Androguard supports automated analysis of multiple applications and integrates with many continuative analysis tools.

The framework of Mann and Starostin [27] targets privacy leaks in Android applications by checking violations against predefined privacy policies. They abstract the Dalvik VM instruction set into smaller groups of instructions and evaluate whether the abstracted instruction sequence corresponds to a known policy. The used policies resemble a set of previously defined sources and sinks of private data in the Android API. Kim et al. [24] have pursued a very similar approach and propose a framework which is capable of running flow-sensitive as well as flow-insensitive

[1] https://code.google.com/p/androguard/

analyses. Both frameworks rely on manually composed lists of sources and sinks and thus might be circumvented. As a countermeasure, Rasthofer, Arzt, and Bodden [39] utilise automated machine-learning for the reliable identication of sources and sinks.

Gibler et al. [18] have demonstrated a framework, suitable for the automated identification of potential data leaks on a large scale. By analysing 24.350 applications within 30 hours, they were able to determine 57.299 potential privacy leaks. In contrast to the previously listed approaches, Gibler et al. [18] apply information-flow analysis on decompiled Java code. First, Dalvik byte-code is reverted to Java code, using the tools ded [31] and dex2jar [2]. Subsequently, the program analysis tool WALA [3] is applied in order to conduct a data flow analysis on the application's call graph. Therein, a set of rules, manually composed by linking Android API methods with Android permissions, is used to to detect leaks. Similarly, the framework SCanDroid by Fuchs, Chaudhuri, and Foster [16] applies WALA in an automated way, to check the security of applications with constraint systems.

In order to achieve high-precision with static taint analysis, Fritz and Arzt [15] have elaborated a framework, called FlowDroid. The novel approach of modeling the entire Android lifecycle, allows most accurate handling of callbacks, asynchronously executing components and multiple entry points. They have implemented a sophisticated concept to cope with aliasing which basically reflects the same strategy that we use in our CryptoSlice tool.

Batyuk et al. [5] propose a Python-based analysis system, which performs static analysis on Smali code in order to detect malicious or unwanted activities. However, the paper is limited to a description of the general framework and grants no insight on the detailed analysis strategy.

For a case study, Enck and Octeau [10] decompiled and statically analysed 1.100 Android applications, looking for vulnerable code or malware. By performing analyses of the control flow, the data flow, the structure and the semantics of applications, they intended to find possible information misuse. They illustratively dissect noteworthy applications and finally reason that many Android APIs need more supervision as they are prone to leak sensitive data.

Moser, Kruegel, and Kirda [29] accentuate the limits of static analysis for malware detection. Being faced with an obscure program flow and binary obfuscation, they demonstrate the need for other analysis techniques.

Applying static analysis on Android applications involves the major disadvantage that Android permits loading dynamic code at runtime, hence making it impossible to be analysed using this approach. Poeplau et al. [36] point out ways to detect code loading techniques and explain how to mitigate possible threats, related to code from external sources. On the other hand, dynamic analysis of additionally loaded code could be circumvented as well, if the code does not match any known signature.

TaintDroid [11] is a framework for dynamic application analysis and runs within a customized Android execution environment. Multiple flows of sensitive data can be tracked simultaneously, while keeping the performance at a high level. It supports real-time monitoring of program behaviour and provides valuable information that facilitate the identification of misbehaving applications. Sarwar et al. [41] evaluate the viability of taint tracking for the detection of privacy data leaks. Furthermore, they propose anti-taint methods, suited to defeat the analysis approach of TaintDroid. The authors conclude that dynamic taint tracking is unlikely to detect privacy leaks if malicious applications actively try to elude analysis.

For the distinction between malicious and benign applications, Alazab et al. [4] use Droid-

[2] https://code.google.com/p/dex2jar/

[3] http://wala.sourceforge.net

Box [4], a dynamic analysis environment, based on TaintDroid. DroidBox enables to run applications in a sandbox and to create graphical reports of suspicious activities. As a result, behavioural patterns can be derived which help to determine families of malware.

Also adapted from TaintDroid, Rastogi, Chen, and Enck [40] have implemented a framework, called Playground. By using an intelligent application execution technique and by triggering sytem events manually, they remedy existing shortcomings of dynamic analysis. The performed framework evaluation with 3.968 applications, revealed leaks of sensitive information in 946 apps.

3.2 Cryptography

With this thesis most closely related is the study of Egele et al. [9], who have statically investigated cryptographic misuse in 15.134 cryptography-related Android applications. In contrast to their work, our tool is not only able to confirm the existence of rule-violating code but we can also pinpoint the origin of problematic statements and appendant invocations. Although their results perfectly underline the problem severity, it is worth taking into consideration that the presented results are in no case reproducible with the given information. For example, the paper makes no explicit statements about the occurrence of multiple ciphers within one application.

Fahl, Harbach, and Oltrogge [12] manually investigate a selection of password manager applications regarding cryptographic aspects and abusable usability features. Due to Android's insufficiently secured clipboard architecture, it was possible to copy-and-paste usernames and passwords and thereby, leak them to sniffing applications. According to the authors, many popular password managers support storing the database at Cloud-based services but fail to use TLS connections appropriately which exposes them to Man-In-The-Middle attacks. In a previously elaborated case study, Fahl et al. [14] demonstrate that their Androguard-based tool MalloDroid detects wrongly applied SSL in 17,28% of 13.500 analysed applications. Related to their findings, Fahl et al. [13] propose countermeasures, such as SSL pinning or different SSL validation strategies. As illustrated by Georgiev et al. [17], broken SSL validation is a fundamental problem in non-browser environments and substantially caused by badly designed APIs. By analysing password managers on Apple iOS and BlackBerry, Belenko and Sklyarov [6] show that problems with weak ciphers or deficient encryption is not a specific problem with Android applications.

In order to impede dictionary and brute-force attacks on cryptographic keys, key derivation functions (KDF), such as PBKDF2 [23] can be used. Adams et al. [2] have presented the most common KDF constructions and shown that the majority has some concerning properties. The study of Abadi and Warinschi [1] focusses on password-based encryption and tries to determine if it is possible for an adversary to make inferences about the password by analysing associated protocol data. Useful recommendations for the practical use of key derivation functions are given by Chen [8]. In particular, security-critical properties, such as the length of the derived key and input parameters are examined.

scrypt is a key derivation function, developed by Percival [35] with intention of making brute-force attacks more difficult. He lists the estimated cost of hardware, needed to break a PBKDF2-derived password in one year. As a countermeasure, he proposes scrypt which increases the security margin by prolonging brute-force attacks significantly. Teufl et al. [44] confirm the improvement in security when scrypt is used. They have calculated the amount of time and costs needed to brute-force scrypt-derived keys using Amazon EC2 Compute Units. It can be observed that the required effort is strongly dependent on the complexity of the underlying password.

[4] https://code.google.com/p/droidbox/

Chapter 4

CryptoSlice - Introduction

This chapter introduces our new static analysis framework, entitled CryptoSlice. Prior to explaining the framework properties, Section 4.1 outlines the objectives of this thesis. Starting with an overview about the main components, the application workflow is illustrated in Section 4.2. We denote the framework features from a user's point of view and explain the individual analysis steps, carried out on each Android application.

In Section 4.3, the required file input format is defined. Assuming that all necessary files are provided, applications have to be prepared before being analyzable. The performed actions are summarized in Section 4.4. Subsequently, we pass on to the actual analysis steps in Section 4.5. We explore the characteristics of each analysis operation and address the produced output.

Finally, Section 4.6 discusses the obtainable analysis results. Accordingly, each step produces results which may be independent from other steps and differ in terms of format and value.

4.1 Objectives

As more and more security-critical applications are used on Android-based smartphones, data integrity has to be ensured by using correct implementations of cryptographic systems. Since software vendors usually provide little information on how responsibly they protect sensitive information, there is an evident need to assess implementations on how they overcome real-world threats.

In this thesis, we aim to provide a way to automatically inspect cryptography-related code in Android applications. We study the most common constructions and distinguish proper usage from misconception. Subsequently, we elaborate a set of inspection procedures suited for application analysis. Since the investigation of cryptographic systems itself is not sufficient to ascertain the integrity of sensitive data, a data flow analysis is required.

Assuming that critical information explicitly concerns user-provided data, it seems reasonable to start tracing the data flow right from that point where it enters an application. In the case of Android applications these trigger points are, for example, input fields for passwords, numeric codes, or phone numbers. The computed paths can then be investigated regarding cryptography-related misconceptions. In order to analyze all usages of certain cryptographic constructions, data flows may also be gathered in backward direction. Within a static analysis environment this approach delivers all paths to parameters which influence the system under investigation.

Evidently, the elaborated security rules are not capable of detecting all kind of weaknesses and, thus, manual analysis using self-definable patterns may be necessary to fill these app-specific analysis gaps. The interpretation of the resulting flows can be facilitated by using an appropriate representation, such as well-arranged graphs. Related to that, it might be convenient to store the obtained data flows on the hard disk for later use in different environments.

After having investigated an application, the result needs to be interpreted. Traditional evaluation approaches tend to categorize an application's behaviour into 'well-behaving' and 'not well-behaving'. However, without regarding the context in which an application is executed, it is barely possible to flag applications as benign or malicious. For instance, not all passwords entered in Android applications require to be processed by cryptographic functions. Depending on the purpose of the application, different security constraints apply. As a consequence, we focus our analysis on detecting and backtracking generally valid security misconceptions rather than drawing conclusions about the overall security of an application. For ambiguous cases, such as password fields, we strive to disclose *potential* security problems. The findings in either case should be saved in a human-readable form while maintaining the possibility for further processing.

Our main objectives can be summarized as follows:

- Elaboration of data flow analysis mechanisms that are capable of tracing information in both forward and backward direction.
- Study of the most common cryptography-related misconceptions in Android applications.
- Practical investigation of applications using previously developed inspection procedures. The expected output are data flow graphs and a list of encountered security problems.

4.2 Overview

The Java-based static analysis framework CryptoSlice is suited to detect and analyse user-definable patterns in Android applications. It provides a holistic environment, capable of carrying out the entire analysis workflow. The actual program investigation is based on a modular approach and, thus, can be extended to serve any particular analysis purpose.

Regarding the objective of this thesis, reliably working static slicing techniques have been elaborated which open up the data flow of relevant code segments. The resulting slices are further processed to automatically identify misconceptions and possible data leak in security-critical code. Therefore, the framework includes a set of security rules which aim to detect and evaluate common flaws in cryptographic code. Assuming that a rule is violated, an XML report is filled with all appendant details, such as the involved class, method, affected code lines, and the reason of violation.

CryptoSlice, furthermore, facilitates the manual analysis of applications by processing user-given slicing criteria. Irrespective of whether data tracking is performed in forward or backward direction, slices can be mapped to well-arranged graphs. Another unique feature is the ability to highlight the track of password fields (or any other resources) throughout the application. The program inspection is separated into multiple components whose concrete architecture deserves a detailed explanation.

Figure 4.1 illustrates the main application workflow. As a first step, an Android application is loaded from a predefined folder. The subsequent preprocessing step is intended to prepare the application for analysis by extracting, decompiling and parsing all appendant Smali files into

an object representation. The following analysis steps actually investigate the previously created application object regarding various aspects. For this thesis, the focus has been put on analysing cryptography-related functions using slicing techniques. Depending on the configuration of the employed analysis modules, the analysis steps end with the XML output of investigation reports. The concluding cleanup step frees the application object from memory and (unless configured differently) purges all temporary files which have been created in the prior decompilation phase.

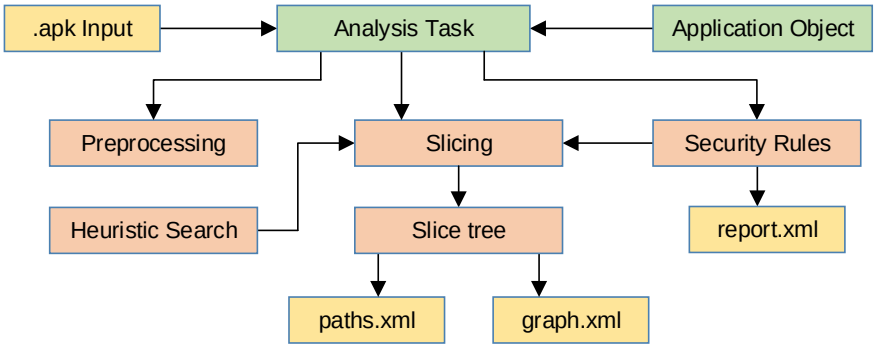

Figure 4.1: CryptoSlice workflow

4.3 Input

The CryptoSlice framework is designed to work with Android application files (`.apk`). As explained in Section 2.2, each of them consists of multiple pieces. The `classes.dex` file comprises the Dalvik bytecode which is transformed to an object representation in the preprocessing step. By analysing the enclosed application resources, the identification of statically defined password fields is achieved. Further data, such as the `AndroidManifest.xml`, provide valuable information about the required permissions and application components. However, for this thesis the manifest file has been disregarded since the exhibited meta-data is not needed in the context of our analysis.

For the investigation of programs at a large scale, our framework provides the ability to automatically process a set of applications, located in the same folder. The availability of multiple CPU cores enables this analysis to be performed multi-threaded which results in a significantly faster execution. In case parallel execution is not applicable, a given set of application is analysed sequentially.

All settings concerning the input processing can be adjusted in a configuration file. Hence, it is not necessary to recompile the framework in order to change the application input folder or to specify the amount of threads to use for parallel execution.

4.4 Preprocessing

First, a given application is dissected and transformed into an object representation. This abstraction contains all relevant information about an application and is used for analysis purposes.

The preprocessing phase is made up of several steps which are executed in consecutive order. Starting with a file integrity check of a given application, the validity of the input file is ensured. Besides, it serves as a quick filter to prevent loading applications without a valid file structure. In the following, a folder structure is initialized for each analysed application which is then used

within all subsequent steps. It retains all application-specific data created during analysis and is recycled in case a program is investigated multiple times.

In the next step, the framework supports unpacking all files from Android applications. Since `.apk` files are in the ZIP format, it is easily possible to extract the Dalvik bytecode and appendant resources. The resulting data is mainly suitable for binary inspection and, thus, is not adequate for the main objective of this thesis.

As a remedy, the subsequent step effectively disassembles the Dalvik bytecode to processable Smali code. Similarly, the manifest file and all resources are decoded in order to render them humanly readable. Since our framework performs static slicing on Smali code, the resources are not immediately involved. However, they are required for the forward tracking of password fields, usually defined in XML format. Both the disassembled bytecode as well as the resources are stored within the previously created folder structure. In case an application is analysed anew, CryptoSlice refers to the already present Smali code and, hence, may skip the now obsolete re-disassembling procedure. Overall, this results in a considerable performance gain. In order to avoid confusion between multiple versions of applications, binary files are identified by their SHA-1 hash instead of using the volatile filename.

After disassembling the Dalvik bytecode to Smali code, the resulting files are parsed into an object representation which is required for the subsequent analysis steps. After having determined the list of available Smali files, CryptoSlice may filter unwanted classes before transforming them into objects. For instance, many applications come with advertisement libraries attached. Processing these libraries would not only waste memory but could also bias analysis results in case the same libraries are used in a multitude of applications. As a consequence, filtering mechanisms ensure that non-relevant files are omitted from being loaded at an early stage. The composed list of Smali files is then parsed into objects which hold all essential attributes and basic blocks. The routine for this process has largely been adopted from the framework SAAF [21], since it already fulfilled the needed operation and only required a manageable amount of extension, optimization, and bugfixing. The resulting list of Smali class objects is suited to be used for the subsequent slicing operations.

4.5 Analysis Steps

Using the previously generated application object, CryptoSlice consecutively executes a series of analysis steps. As depicted in Figure 4.2, the investigation process starts with a *Heuristic Search* operation. The subsequent *Slicing* step is not dependent on the results of the preceding step although the search results might potentially influence the slicing pattern. Under normal circumstances, however, the slicing criterion is either loaded from a configuration file, or defined within a security rule. During the slicing process, the findings are instantly added to a graph representation which, in the end, models all data flows between the slicing criterion and traced endpoints. Based on the *generated graph* the slicing results can be further evaluated and interpreted.

4.5.1 Heuristic Search

The Heuristic Search is capable of identifying predefined events, such as the use of specific cryptographic libraries, or custom SSL trust managers. Using the information gained in this step, the subsequently employed slicing patterns can be adjusted to better fit the currently analysed target. For example, in case the heuristic search step reveals that an application is not using the BouncyCastle library at all, slicing operations on associated library methods may be omitted.

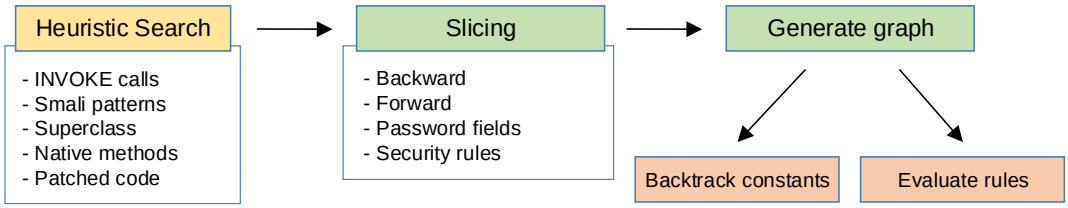

Figure 4.2: Analysis steps overview

Moreover, each analysis process is extended with meta-information about security-related properties. For instance, it is possible to determine the use of native code and classloaders. While native code cannot be tracked by our slicer, the search for classloaders potentially discloses that additional Dalvik bytecode is loaded from external resources. In both cases, it is indispensable to give advices that manual investigation is needed.

Related to that, determining the invocation of certain methods is also advantageous to quickly assign applications to preselected categories. For example, finding methods related to SSL connections may indicate that an application is basically designed for the protected exchange of data. Considered by itself, this statement still requires further investigation in order to clarify the kind of potentially securely transmitted data. Beyond that it also gives reason to verify the parameters of the underlying SSL implementation, such as the use of the SSL trust manager and hostname verifier.

A similar feature is the detection of XML resources usage throughout an application. Firstly, it is possible to find out whether a specific type of resource is employed, e.g. input fields for passwords, phone numbers, or PIN codes. In a second step the gained knowledge can be used to actually discover concrete usage within the application. As a result, slicing criteria can be defined on-the-fly, serving as input to the subsequent slicing process. In this scenario, the Heuristic Search functionality is intended to dynamically align the analysis process according to the current application. Nevertheless, it also delivers valuable information about the substance of a target. For example, it is feasible to rapidly inspect a given set of applications regarding the combined use of password fields and HTTP- or HTTPS-related methods. The overall objective, of course, is to evaluate whether sensible information could possibly be leaked.

4.5.2 Slicing

The Slicing functionality of our framework tracks a given pattern and stores the resulting data flows in an object-based graph representation. The process can be conducted in both forward and backward direction. We use program slicing mainly to isolate those parts of a program that are relevant for a specific slicing criterion. In a broader sense, however, we are able to use the technique in order to answer the following questions, drafted in plain language:

- What happens with a password entered in the application?
- Is common cryptography applied in a secure way?
- Does the application consider common security practices?
- Is the application likely to leak sensitive data?

As a result, the slicing process delivers a data flow which models the path from the slicing criterion to one or more endpoints. In a further step, the obtained results can be analysed and evaluated regarding a specific analysis purpose. In the light of the previously listed questions, slicing is used as a means to an end rather than exposing concrete answers on its own.

Forward Direction

Forward slicing aims to discover the program subset that is very likely to be affected by a given slicing criterion. In other words, it can be used to find out those parts of an application that depend on a particular statement. In our case, this technique is employed to track any kind of resources, such as text and password fields, but also to follow a specific object in forward direction.

First, our framework requires a slicing pattern to be defined. Specified in XML format, it conceptionally describes the type of resource or object that should be tracked. Since a pattern includes no reference to a specific program statement, it does not represent a slicing criterion by itself (see Section 2.6.1). Instead, it contains all necessary data to dynamically build slicing criteria corresponding to the pattern. Assuming that the data of interest occurs multiple times within an application, a multitude of slicing criteria is deduced and subsequently tracked. Depending on the defined focus and level of granularity, a pattern might be applicable to either only one specific application, or be generally suited for a large set of targets.

As depicted in Listing 4.1, XML resource elements of Android applications can be tracked by describing them as XPath queries. The advantage of this approach is that basically any kind of resource type and attribute combination can be declared within the same pattern structure. The given example intends to track all uses of `EditText` elements which have an attribute `inputType` with the value `textPassword`. In the Android context this pattern generally describes password input fields that are shown to the user during normal interaction with an application.

```
1 <forwardtracking-pattern enabled="true" type="XPATH_QUERY"
2     pattern="//EditText[contains(@inputType, 'textPassword')]"
3     description="EditText fields with inputType containing textPassword" />
```

Listing 4.1: Forward slicing pattern: Password fields

Similarly, arbitrary objects can be tracked in forward direction by specifying a method and the corresponding class. Based on the given type signature, matching object instances are looked up in the Smali code. For each instance, the subsequent slicing operation models a data flow until the track is lost or spurious. Listing 4.2 illustrates the definition of a pattern for all object instances of `MessageDigest` classes. The method `getInstance()` returns a static instance which is used as a slicing criterion. In other scenarios, instances are created by calling the constructor. To track these instances from the time when they are constructed, the pattern method has to be set to `<init>`. Although any method can be specified, in principle, it seems reasonable to start at the constructor in order to capture all program statements that are influenced by a particular object instance.

```
1 <forwardtracking-pattern enabled="true" type="OBJECT"
2     class="java/security/MessageDigest" method="getInstance"
3     description="MessageDigest object" />
```

Listing 4.2: Forward slicing pattern: MessageDigest->getInstance() objects

Backward Direction

Backward Slicing, as an opposite to forward slicing, intends to find the program subset which has influenced a given slicing criterion at a certain point. Using static backtracking we are able to determine all program statements on which a given criterion depends. In our framework, the technique is used to track method parameters in backward direction until the retrieval of one or more constant values terminates the slicing process.

In analogy to the aforementioned XML format, a similar description has been elaborated for backward slicing patterns. As demonstrated in Listing 4.3, compatible patterns have to include the particular method of interest, alongside with the corresponding class and method parameters. Since we are applying the slicing process to Smali code, the method signature needs to be written in Dalvik notation (see Table 2.1). Lastly, the index of the parameter that should be tracked, is defined by the attribute `interestingParameter`. In the given example, index 1 stands for the second method parameter `[B`. As a whole, the given pattern serves to track the second parameter of each constructor invocation of the class `PBEKeySpec` that matches the method signature `[C[BII`. Assuming that the method of interest is overloaded, the signature might optionally be set to `*` in order to always track the indicated parameter index, irrespective of the actual signature.

```
1 <backtracking-pattern enabled="true"
2     class="javax/crypto/spec/PBEKeySpec" method="<init>"
3     parameters="[C[BII"
4     description="PBEKeySpec: salt" interestingParameter="1" />
```

Listing 4.3: Backtracking pattern: PBEKeySpec salt

When started, the slicing process first determines all invocations of the specified method and dynamically builds slicing criteria corresponding to the given pattern. Subsequently, for each criterion the register is identified which matches the defined parameter of interest. While tracking the register in backward direction, all involved program statements are recorded as slice nodes and added to a slice tree. The backward tracking ends when constants are assigned to the currently tracked register and slicing is no longer feasible. Likewise, the slicing process terminates when the tracked register is overwritten or track is lost, for example, when referenced methods cannot be resolved.

Slice Trees and Constants

The slicing process dynamically builds a tree with all encountered slice nodes for a specific slicing criterion. The top node is always the criterion, deduced from the pattern, since it represents the root of all possible execution paths that can be modeled. Subjacent nodes stand for all code lines which are contained in the slice. In case there are multiple execution paths (e.g. if-else statement), a slice node might have links from multiple predecessor nodes. When code statements are iterated multiple times (e.g. for or while loop) cycles are induced between vertices. Each (intermediate) node involves a list of all predecessor nodes, including the originating register name and the register name, related to the current program statement.

A slice tree can comprise one or multiple leaf nodes whereas each describes either a constant or indicates an abruptly ended slicing process. Assuming that a constant value, such as an integer, an array, or a string, is copied into the tracked register, slicing may stop since the register value is redefined. For backward slicing this signifies that the tracking process has led to one or more values that affect the slicing criterion. In contrast, for forward slicing it means that the currently

tracked register will not affect any subsequent operation and, thus, the data flow has reached an endpoint. Leaf nodes are also inserted in case slicing loses track. This happens, for instance, when registers are set as parameters in calls to unresolvable methods.

Since all disclosed path endpoints are invariant, we regard them as constants. Aside from containing information about values that are assigned to registers, constants also explain why paths end at certain points. This is achieved by retaining metadata from slicing. For example, each constant is assigned a category which clearly defines the type of the underlying value. Similarly, in case tracking stops abruptly, constants are put in place to describe the cause. Using the available metadata, it can be highlighted whether slicing ended due to a native library call, an unknown API method, an event-triggered method (callback), or for other reasons.

As already mentioned, the tracking process emits a tree for each slicing criterion. Since a user-defined pattern can lead to multiple slicing criteria, the tracking process terminates by collecting all constructed slicing criteria with the according slice tree. The overall result for a particular pattern is suited for subsequent evaluation purposes. Analysing the results instantly during slicing has been disregarded since the trees are likely to change and be extended continuously.

4.5.3 Backtracking Constants

Having built the slice trees for a specific pattern, our framework supports extracting individual paths from a slicing criterion to the constants found. Technically, for each constant all possible data flows up to the top of the graph are traversed and segregated. Depending on the amount of paths between an endpoint and the slicing criterion, a plenitude of traces emerge. Overall, they are modelling all feasible execution paths for a tracked variable to reach a particular endpoint.

At the beginning, all slice nodes with no successors (leaves) are identified. Subsequently, the slice graph is searched for all unique paths leading to the slicing criterion. Since iterated code statements may have led to cycles during slicing, our traversal algorithm needs to take them into account in order to avoid a path construction ad infinitum. The resulting paths correspond to the flows that are contained in the slice graph as well. Disassociated paths for each constant, however, may be better suited for analysis purposes that need individual flows without cycles and branches. In our framework, we are reliant on this feature within the scope of the elaborated security rules (see Section 4.5.4). By checking paths for specific criteria, we can draw conclusions about an application's behaviour and probable misconfiguration.

Aside from using the paths internally, they can be exported in XML format for further processing and manual analysis. Being prepared as a stacktrace-like sequence, each node of the unfolded path shows the class, method, and line number corresponding to the tracked code statement. The human-readable result is suited for manual analysis but can also serve as a basis for further data flow processing. For example, data flow graphs could be abstracted by identifying common subgraphs or by matching similarities using probabilistic methods.

4.5.4 Security Rules

For the detection and analysis of security-related misconceptions, the CryptoSlice framework includes 9 security rules. Leveraging the previously described features, the rules are designed to identify and highlight common implementation weaknesses and flawed security properties. However, their objective is not to detect and analyze *all* sorts of security-related code and, in particular, not to track down custom implementations of cryptography. Instead, they aim to verify the correct use of standard APIs intended to protect sensitive information.

Before switching to self-designed data protection mechanisms, developers are likely to recall functionality that is already provided. On Android, security-related capabilities are exposed by the Java Cryptographic Architecture (see Section 2.7). While modern development environments give advice on general security attributes, such as preferring HTTPS to HTTP, they are currently unable to draw conclusions on cryptography-related code. Similarly, official API documentation lacks information on how to choose adequate parameters. For instance, although it is denoted that ECB mode should not be used for multiple blocks of data, the citation of possible implications and better suited alternatives is omitted. Related to that, default properties might be inappropriate in a particular context. For example, when specifying AES for encryption (without any further arguments), the security provider implicitly chooses ECB mode.

Each of the implemented rules first tries to find predefined method signatures and then, based on the found matches, verifies whether they correspond to the predefined security model. In case a rule is violated, an alert is issued, accompanied by details about the problematic statement. Thereby, the framework manages to evaluate the exact location of rule violating code lines and writes information about the involved class, method and code line to the console as well as to an XML report. As a consequence, the result is easily readable and can, moreover, be interpreted automatically. On a larger scale, it enables rules to be applied to a bulk of applications without requiring interaction.

The security rules are executed in consecutive order and independent from each other. The only shared feature is a reporting system, used to summarize the output of all evaluated checks. Each rule incorporates the entire workflow described in the previous sections. Due to the extensible approach, adding further security rules is supported by requiring only little effort.

According to best practices for the secure application of cryptography and protection of sensitive data, a set of rules has been elaborated, uncovering typical misuses:

1. No ECB mode for encryption
2. No non-random IV for CBC encryption
3. No constant encryption keys
4. No constant passwords or salts for PBE
5. Not fewer than 1000 iterations for PBE
6. No static seeds for SecureRandom()
7. No MessageDigest without content
8. No password leaks

While the first 6 rules have been adopted from Egele et al. [9], our tool pursues a different analysis approach which supports pinpointing the exact origin of problematic statements in Smali code. In addition, the methodology of the proposed rules has been refined in terms of accuracy and reproducibility. The remaining rules are entirely new contributions that specifically leverage the capabilities of our slicer and enable to cover a broader analysis scope.

Intentionally, not all rules are targeting exclusively cryptography-related aspects. For instance, one rule declares that user-entered passwords must not be passed over to non-adequate methods. Although the data flow of password inputs may involve cryptography, its pertinency depends on the specific use case of a password. In other words, not all passwords are required to be subject to cryptographic transformation. As a consequence, all rules are directed to disclose generally valid misconceptions, independent of a particular deployment scenario. A detailed description and risk analysis of all implemented rules is provided in Chapter 6.

4.6 Results

Since all analysis steps investigate different aspects, no generalized conclusions about the security of an application can be drawn without further explanation. In order to retain all analysis results such that they are usable for further processing, XML reports are emitted for each application.

Within the heuristic search step the invocation and presence of Smali code, native code, libraries, etc. is determined. Assuming that the searched patterns influence the subsequent slicing operation, the results from slicing implicitly contain the output from heuristic search. As a consequence, it has been disregarded to print the findings separately.

The slicing step models all possible execution paths for a tracked variable in forward or backward direction and, in addition, composes a list of constants (see Section 4.5.2). Represented as a slice tree, the data can either be graphically visualized for manual analysis, or stored in XML format. In the first case, a tree is plotted in PDF format using graphviz [1]. The resulting graph illustrates all data flows between the slicing criterion and traced endpoints. In the other case, paths are broken down into sequences, each ranging from the corresponding slicing criterion to a path endpoint, and output alongside the appendant slicing pattern. Either way, slicing results need to be further processed in order to generate a benefit.

The findings and conclusions of all security rules are added to a separate XML report. Each evaluated rule integrates its own child element with an individual sub-tree structure. A sample output for the rule *No constant passwords or salts for PBE* is provided in Listing 4.4, indicating the use of a fixed salt value for password-based encryption. Aside from highlighting the specific salt value, the code line of the appendant `PBEKeySpec` invocation (in Smali code) and the containing method signature are listed. In case an application violates the rule multiple times, each case of non-compliance is denoted within an individual `pbeKeySpec` block.

```
1  <cryptoRule number="4" title="No constant passwords or salts for PBE">
2    <pbeKeySpec>
3      <method>package.d.v: private static a([C[B)V</method>
4      <codeline>123</codeline>
5      <constantSalt>
6          <type>String</type>
7          <value>sryL3uJxSG1)ViA[vH3XX0u+uKS3bdMf</value>
8      </constantSalt>
9    </pbeKeySpec>
10 </cryptoRule>
```

Listing 4.4: XML output: No constant passwords or salts for PBE

The overall results provide a valuable contribution to Android application analysis. By processing the data flow paths from slicing, conclusions might be drawn about a program's behaviour.

[1] http://www.graphviz.org

Chapter 5

Static Slicing of Smali Code

The ability to trace information in both forward and backward direction is a core component of CryptoSlice. This chapter presents the implemented techniques for static slicing and highlights practical challenges. First, Section 5.1 briefly introduces the basic environment. Section 5.2 explains how slicing criteria are derived from user-provided slicing patterns. In Section 5.3, we elaborate approaches to derive slicing criteria when tracking password input fields in forward direction. Starting with password fields in XML resources, we also address input fields that are dynamically generated during runtime.

Section 5.4.2 illustrates the employed strategy for backward slicing, while Section 5.4.3 outlines a method to perform forward slicing on Smali code. Section 5.5 introduces the concept of a dynamically built tree which retains all information from a slicing process. Based on the generated graph, we are able to visualize information flow dependencies and to perform further analysis.

Finally, Section 5.6 gives an understanding of the limitations of our implementation. We disclose possible implications and outline their relevance in real-world deployment scenarios.

5.1 Introduction

By performing static slicing on Smali code, our framework is capable of determining the control and data flow of relevant code segments in Android applications. Based on a given pattern, an analysis is conducted in forward or backward direction, storing the results in an object-based graph representation for further inspection. In the following, we introduce the basic environment and outline essential functionality.

Having derived one or multiple slicing criteria from a given pattern, they are initially added to an internal FIFO queue. As depicted in Figure 5.1, this to-do list serves as input for both the forward and backward slicer and collects all registers, fields, return values, and arrays that are subject to tracking. Moreover, it holds a reference to all objects that have already been followed and excludes them from being reprocessed. When requested by the slicer, the queue returns the next object to track, which includes the register to track and the location of the corresponding opcode. On the basis of this approach we are effectively able to control the slicing process and prevent the repeated analysis of already investigated data flows.

Forward and backward slicing are conceptually separated components that process the input from the to-do list and output slicing results to a dynamically built tree. Initially, the slicing criterion is set as root node, followed by all code statements that are contained in the slice. The generated graph can be used for further analysis and evaluation purposes.

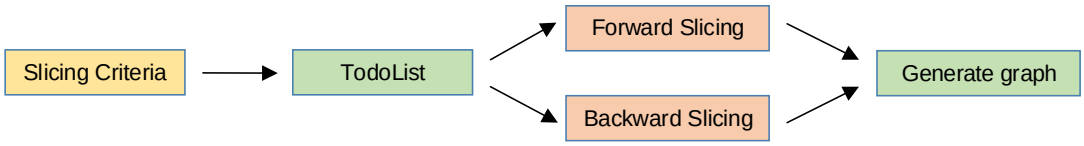

Figure 5.1: Static Slicing process

5.1.1 Slicing Accuracy

The aforementioned queue ascertains accurate analysis results by filtering registers that exceed a predefined threshold of fuzziness. Each tracked register is assigned a *fuzzy level* which indicates its accuracy in accordance with the slicing criterion. In other words, it expresses the likelihood that the value of the currently tracked register still equals the value of the initial register. Accordingly, the fuzzy level is also attached to found constants and nodes within the slice tree in order to highlight their relevance with respect to the slicing criterion. A value of 0 means that the result is completely accurate and has not been modified on its way to the slicing criterion. Higher values indicate less accurate results and a reduced expressiveness of the results.

An example of a fuzzy backtracking search is illustrated in Listing 5.1. Assuming the slicing criterion $C = (9, \{v2\})$, initially the code statement at line 9 is added to the slice. Subsequently, all preceding code statements are traced, that have an influence on the register value of `v2`. While tracking this register, the slicing process encounters an invocation of `toCharArray(v1)` at line 7. Resolving the method is not feasible as it points to a Java library method. As a consequence, the slicer *blindly assumes* that the passed register `v1` affects the resulting register `v2` and elevates the fuzzy level of `v1` before adding the register to the to-do list. Subsequently tracked registers always inherit the fuzzy level of the register they are derived from. Since the same procedure also applies to the unresolvable methods in line 5 and 3, the finally found constant value in the first line is assigned a fuzzy level value of 3.

```
1 const-string v0, "Sample string"
2 new-instance v1, Ljava/lang/StringBuilder;
3 invoke-direct {v1,v0},Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V
4 move-result-object v1
5 invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
6 move-result-object v1
7 invoke-virtual {v1}, Ljava/lang/String;->toCharArray()[C
8 move-result-object v2
9 invoke-direct {v2}, Ljavax/crypto/spec/PBEKeySpec;-><init>([C)V
```

Listing 5.1: Fuzzy backtracking example

Although the fuzzy level enables us to measure uncertainty in analysis results, it makes no indications about the quality of found constants. For example, a high value does not necessarily imply that a constant has only marginal impact on a slicing criterion. Similarly, it is probable that a register has a low value but does not correlate with the initial register at all.

5.2 Slicing Patterns

As already described in Section 4.5.2, the slicing process naturally depends on a slicing criterion. Since our objective is to perform an automated analysis on any kind of provided slicing pattern, it contains no reference to specific program statements. As a consequence, slicing criteria have to be derived dynamically for each application under investigation.

The slicing process starts by processing a given slicing pattern. In order to comply with different requirements, patterns support the description of different types or features that can be tracked.

Method invocations

In the default behaviour, slicing criteria are determined by searching for all `invoke` statements matching the given pattern. Therefore, all code lines of an application are scanned, looking for the provided method signature. For each matching invocation, the appendant program statement is considered as a starting point for slicing. Subsequently, the name of the register to track is located by associating the index of each occurring register with the given parameter (index) of interest. As a result, a set of suitable slicing criteria is delivered.

Return values (in backward direction)

Method invocations can only be tracked if a method is actually called from within the available application context. However, in case an application declares a method but invokes it only from the outside, e.g. using the Android API or a native library, the previous approach is insufficient. As a remedy, it is possible to perform *backward slicing* in order to find out all statements that affect the method's return value. Tracking input parameters and return values of the concerned method in forward direction is unfeasible, since the invocation goes beyond the scope of the analyzed application. The described functionality is intended to track methods overridden from a base class with no apparent invocation. Practically, a slicing criterion is built for each return statement of the specific method, defined by the corresponding code line and register name.

Specific resource IDs

Android applications usually include resources, such as user interface layouts and strings, that are externalized from the program code. For every outsourced element, a developer has to assign a unique resource ID which can be referenced in code by using the syntax `R.<resource_type>.<resource_name>`[1]. When an application is packaged, the compiler automatically substitutes each reference to a resource in code by a randomly derived integer value. In addition, it generates a class `R` which contains a list of all available resource IDs and establishes a mapping between IDs and corresponding integer values.

For example, assuming that a resource is labeled `@+id/etUsername1`, it can be accessed from code using the variable name `R.id.etUsername1`. When the application is compiled, each reference to the resource is replaced by a distinct static integer (e.g. `0x7f0a0004`). The mapping of this value to the related resource ID is finally stored in the `R` class.

In order to track particular resources in Smali code, in a first step it is required to find out the integer constant linked to a given resource ID. According to the specified resource type, the proper subclass of `R` is looked up and checked for a suitable mapping. Since there are multiple

[1] https://developer.android.com/guide/topics/resources/accessing-resources.html#ResourcesFromCode

possibilities how the relation can be structured, the search strategy adapts for different conditions. Prior to determining the associated value, it is ascertained that a subclass of `R` contains a member with the particular resource ID. Despite the fact that a value is always assigned statically to the member, it does not necessarily have to be a constant. Hence, we look for both *immutable* (static final) and *mutable* (only static) assignments. In the former case, the searched integer is known at compile time and can be found alongside the field definition. In the latter case, it is comprised within a static initializer method, previously added by the compiler in order to define field values when the class is first accessed. Thereby, the constant value can be determined by backtracking the `sput` instruction, ultimately used to assign it to the field.

In some cases, Android expects resources to be linked with deterministic integer values. Since the previously described R subclass is re-generated every time an application is compiled, fixed resource references have to be stored in a separate file, named `public.xml`. Similar to subclasses of `R`, entries of this file have to be considered as well when tracking specific resource IDs. In either case, if a given resource ID is contained, the outcome is a static integer which is used by the program when referring to resources.

As a second step, resource usage is determined by searching the static integer in all code lines of an application. For each `const` instruction that assigns the integer value to a register, we assume that subsequently executed code is going to with the underlying resource. The same principle applies to all fields which declare the value immutably. Consequently, all matching constant definitions are translated to slicing criteria, whereas each represents an individual access to a particular resource. Since a program cannot make use of a resource until assigning the corresponding integer value to a register, the inferred slicing criteria are only suitable for forward slicing.

Arbitrary resource objects

The main drawback of tracking specific resources is that the employed slicing patterns are customized for particular programs. Since resources are usually labelled by the developer, slicing patterns with concrete resource IDs are unlikely to yield results, when applied to random applications. While the manual selection of relevant resource IDs appears to be very tedious, tracking all of them would also evoke superfluous slices. As a remedy, CryptoSlice supports the generalized description of resources using XPath queries.

Based on the fact that Android resources are typically denoted in XML format, XPath queries are suited to select elements by means of their node type and a variety of predicates. Accordingly, it is feasible to assemble slicing patterns that focus on particular resources in a multitude of applications. In contrast to tracking specific resource IDs, the used XPath queries are intended to cover one or multiple resource elements. By leveraging the flexibility of XPath, queries can be adapted to select arbitrary resource elements that match given properties. In practice, this benefit enables slicing patterns to be generalized to such an extent that the characteristics of individual applications become entirely extraneous.

As explained, the most prominent advantage of identifying resources by meta-information, instead of a specific resource ID, is the fact that slicing patterns are suited to target multiple resources within one application. While this behaviour is definitely a desired objective, it inherently leads to the tracking of resources that might not be relevant within the context of the conducted analysis. For example, when tracking all input fields for numeric values (e.g. `EditText` elements with the attribute `inputType` having the value `number`), no conclusions can be drawn about the purpose of the emerging slicing criteria. A more precise XPath query, however, can prevent the selection and subsequent tracking of unrelated resources.

5.3 Tracking Password Fields

The analysis of data flows from input fields for passwords starts with the definition of a suitable slicing pattern. Based on the provided parameters, concrete password field usages are searched in program code, added to slicing criteria and can then be tracked in forward direction. In view of our analysis objectives, the following case study illustrates the derivation of an eligible pattern. With the intention of tracking any password field occurring in practice, we also identify possible shortcomings of an elaborated pattern.

Basically, password fields in Android applications are either statically defined as XML resources or generated from program code during runtime. Since both options refer to the same implementation internally, their capabilities and produced outputs are identical. As an initial trigger for slicing, however, it is not feasible to cover both forms by a single slicing pattern. This is also reflected by our slicing patterns' types which focus either on resource objects or statement invocations. In the following, we will examine both cases and highlight their characteristics.

5.3.1 XML Resources

Password input fields in XML resources typically make use of the element class `EditText` that enables editable input fields to be displayed. Depending on the provided attributes, differently shaped fields and keyboards are presented to the user during interaction. Concise XPath queries facilitate the selection of corresponding input fields for analysis purposes.

Until the release of Android 1.6 (API level 4), the default way to declare password input fields consisted in adding the property `password=true` to an `EditText` element. Although considered deprecated now, the technique can still be found in applications that maintain compatibility with the eldest versions of Android. Referring to the previous section, an XPath statement is suited to specifically match this password input field description. The first-mentioned slicing pattern in Listing 5.2 illustrates the assembled XPath query.

On current versions of Android, password fields are declared by setting a corresponding constant value to the `EditText` element property `inputType`. Alongside with other input types, the change also introduced more fine-grained descriptors for password input fields. For instance, developers can specify the type `numberPassword` in order to restrict possible user input to numerical values only. For the subsequent static slicing process, this implies that the initially tracked value is also numeric and, hence, likely to be subject to integer transformations. If the property `maxLength` is also set, conclusions about the achievable security grade could be drawn even without slicing.

The most obvious descriptor for an arbitrary password combination is the input type value `textPassword`. Considering the previously formulated pattern, the same scheme is applicable to the input type property. The resulting adaptation is depicted in Listing 5.2. In the current state the XPath statement is designed to match *exactly* the given predicate and fail for any deviation. Although it is suited for practical application, the precision is comparably low as other relevant and legitimate input type values are not taken into account. In particular, this concerns all other descriptors, designated for password input, such as `textWebPassword`, `textVisiblePassword`, and `numberPassword`. A possible remedy is to add the listed options to the XPath statement accordingly. The resulting query is now capable of delivering all elements where there is an exact match in terms of input type value.

Another possible application scenario is the combined use of multiple input types. The value `textNoSuggestions|textPassword`, for example, causes the user-shown keyboard to omit the display of any dictionary-based word suggestions. Without adaptation to this circumstance our

XPath query would not match input type combinations at all. A pragmatical approach to this issue consists in refining the pattern in a way that it focusses on verifying the occurrence of a password type, disregarding further options. This can be achieved by simply checking whether the property *contains* a known value. In contrast to the previously stipulated exact conformity, we weaken the statement to a containing match. The final slicing pattern is denoted in Listing 5.2. It covers all relevant forms of password types and, at the same time, refrains from matching unrelated values.

```
<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
    pattern="//EditText[@password='true']"
    description="EditText XML fields with attribute 'password'" />

<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
    pattern="//EditText[@inputType, 'textPassword']"
    description="EditText fields with inputType = textPassword" />

<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
    pattern="//EditText[contains(@inputType, 'textPassword')
      or contains(@inputType, 'textWebPassword')"
      or contains(@inputType, 'textVisiblePassword')"
      or contains(@inputType, 'numberPassword')]
    description="EditText fields with password inputType" />
```

Listing 5.2: Forward slicing pattern: Password fields (pattern elaboration)

Using the elaborated pattern for the tracking of password fields, our framework is capable of identifying all matching resources within an application. The general workflow for this analysis basically extends the concept of tracking arbitrary resources, described in the previous section. In this respect it has to be pointed out that the analysis process always targets the resource object rather than any of its attributes. In the current context this means that not the user-entered password value itself is tracked but the object enclosing it. When the resource is accessed in Smali code, the resulting data flow might include routines (among others) that process password values. Consequently, the delivered slicing results are obviously subject to a certain extent of imprecision.

5.3.2 Generated Input Fields

Another possibility to display password fields is to generate them dynamically during runtime. Rather than embedding monolithic `EditText` elements in XML resources, editable fields can also be defined using program code. Accordingly, a variety of properties and actions is assignable on each instance of the class `EditText`. A slicing pattern should, hence, be suited to identify generated password fields reliably and to convey slicing criteria for the subsequent tracking process. In order to achieve this, we have to cope with three essential problems:

- How is it possible to distinguish between ordinary `EditText` elements and those that are configured for password input?
- What are the implications of tracking the entire element instead of the password value only?
- Are we able to design an appropriate slicing pattern that adapts to the given constraints?

These questions were equally relevant for password fields in XML resources. Nevertheless, in the former case it has shown to be fairly simple to derive a pattern that matches particular properties of one corresponding XML element. With generated input fields, more complex prerequisites apply

since password fields cannot be reduced to a single program statement, enclosing all relevant attributes. In the following, we will gradually answer the previously listed questions by examining the sample code provided in Listing 5.3.

Password Field Identification

Initialized within the corresponding application context, a dynamically created input field is an instance of the class `EditText`. In order to hide the user-entered text by asterisks, an input field has to be assigned an appropriate *password transformation method*. Similar to XML resources, an optionally added *input type property* restricts the possible input value to a predefined set of characters and advises the keyboard not to save the password for spelling correction. Although not recommended from a security-aware perspective, specifying the input type may be omitted. Consequently, we can conclude that the only irrevocable indicator for a password field (with asterisks) is the assignment of a `PasswordTransformationMethod` class instance. In order to identify an employed transformation object and input type constant, the arguments of the corresponding methods `setTransformationMethod` and `setInputType` have to be tracked backwards.

With visible (non-hidden) password input fields, an entered text undergoes no transformation and, hence, in that case the value of the input type property remains the sole indicator for a password input field. As illustrated in Listing 5.3, the type is declared by a constant value which first points to the possible user-entered values (e.g. text or number) and secondly specifies the particular type of the input field. Accordingly, for visible passwords the second descriptor would be `TYPE_TEXT_VARIATION_VISIBLE_PASSWORD`. The constant states whether an input field is designed to handle passwords and, moreover, indicates the processed data type.

Being assembled at runtime, it might occur that the input type is not immediately assigned to the `EditText` instance upon initialization. Similarly, it is probable that the transformation method changes during execution. This is likely the case with Android applications that offer users the option to toggle the password visibility by clicking on a button. Internally, this is achieved by switching the transformation method, e.g. from `PasswordTransformationMethod` to `HideReturnsTransformationMethod` (or any other non-hiding option) and vice-versa. Unless the password transformation is already registered upon initialization, it is evident that *all* transformation method assignments to an `EditText` instance need to be backtracked in order to determine whether the element acts as an input for passwords at any point of execution. Of course, this process becomes redundant and can be skipped if an input type is set, already referring to a password.

Overall, the workflow to find generated password input fields can be summarized as follows:

1. Find instances of `EditText` objects and, using forward slicing, verify whether the methods `setTransformationMethod` and `setInputType` are invoked directly upon initialization.
2. Based on the obtained results, backtrack the arguments passed to the found methods. An input field for passwords is found if at least one of the following conditions is met:
   (a) The tracked transformation method corresponds to an instance of the class `PasswordTransformationMethod`.
   (b) The tracked input type constant value indicates a matching field type for a visible, numeric, web, or general password.
3. If still undecided, track all transformation method or input type assignments appendant to a particular `EditText` instance and perform the evaluation as outlined in the previous step.

```
1  AlertDialog.Builder alert = new AlertDialog.Builder(context);
2
3  final EditText input = new EditText(context);
4  input.setTransformationMethod(PasswordTransformationMethod.getInstance());
5  input.setInputType(InputType.TYPE_CLASS_TEXT | InputType.
       TYPE_TEXT_VARIATION_PASSWORD);
6
7  input.addTextChangedListener(new TextWatcher() {
8    @Override
9    public void onTextChanged(CharSequence s, int st, int before, int ct) {
10     String password = s.toString();
11   }
12
13   @Override
14   public void beforeTextChanged(CharSequence s, int st, int ct, int af) {}
15
16   @Override
17   public void afterTextChanged(Editable s) {
18     String password = s.toString();
19   }
20 });
21 alert.setView(input);
22
23 Button submitButton = new Button(this);
24 button.setText("Submit");
25 button.setOnClickListener(new View.OnClickListener() {
26   public void onClick(View view) {
27     String password = input.getText().toString();
28   }
29 });
```

Listing 5.3: Dynamically generated input field

Tracking Passwords

Having successfully identified an `EditText` element as a container for password input, the subsequent task consists in tracking the data flow of a user-entered password. Beforehand, a suitable slicing criterion is needed in order to trigger this process. In the following, we will highlight the available options and point out possible implications on the slicing results.

Basically, it is conceivable to compose a criterion from the previously found `EditText` instance and track the object in forward direction. The resulting slice would, in theory, comprise all code statements that refer to the input field or any of its properties. Applied to the sample code provided in Listing 5.3, the result *should* include the code lines 10, 18, 21, and 27 since they reference the input field object or a derivative. However, as opposed to the directly visible data flow from the `EditText` instance to the `AlertDialog` in line 21, the affiliation with the other code lines is not immediately obvious. In order to find these traces, a slicer has to be aware of *implicit control flows*, internally handled by the Android framework.

As depicted in Listing 5.3, `EditText` objects support the registration of event-triggered methods. They enable a predefined callback to be invoked whenever the event is signaled. The sample code demonstrates this feature by means of the `addTextChangedListener` listener. In practice, it causes the method `onTextChanged` (line 9) to be called with the current input field text wrapped as a `CharSequence`, as soon as the text of the input field changes. Another listener method is

attached to a button (line 25) and brings the method `onClick` to access the value of the input field (line 27), once the button is clicked. The actual control and data flow in these two examples is carried out internally and beyond the scope of the underlying program code. For static slicing, this means that neither a consecutive nor a coherent data flow is determinable due to missing links in the execution chain. For instance, without being able to track into Android's `TextWatcher` class, a slicer cannot know that the `CharSequence` encloses the value of the input field. More generally, the slicer will miss all information flows that are handled within a listener-callback system, leading to considerable imprecision.

One way to address the shown issue consists in statically linking callbacks and their registrations. For instance, assuming that a call to `addTextChangedListener` is encountered by the slicing process, a previously learned mapping could disclose that the actual input value is made available through a `CharSequence` or `Editable` parameter. The downside of this approach, however, is that all probable associations have to be known in advance. Considering the extensive amount of event listeners and possible callbacks on the Android ecosystem, a manually managed database is likely to cover only a subset of all possible implicit control flows.

Instead of tracking `EditText` instances, another approach is to track methods that are known to access the password value. For example, by defining invocations of `EditText->getText()` as slicing criterion for forward tracking, it can safely be assumed that the initially sliced register holds the actual password value. Employing the same criterion for backward slicing reveals whether the originating `EditText` instance sets an appropriate transformation method or input type. Compared to the formerly described method, this combination of slicing into both directions enables the resulting slice to start with the password value itself (instead of the input field) and ascertains that it is not influenced by unrelated properties of the originating `EditText` object. However, the focus on specific methods, such as `getText()`, also causes other accessors to be excluded a priori.

The following key points can be concluded from the described approaches:

- The slicing criterion has to be assigned an `EditText` element or an access method, such as `getText()`, in order to track password input fields.
- Depending on the initial trigger, the slicing results may include code statements that are not related to the input field value at all.
- User-entered passwords are typically passed to event-triggered callbacks. Since the slicer cannot reproduce this information flow, it is not resembled in the slicing results.
- By implicitly referring to an `EditText` instance, password values are made available via different data types and access descriptors. The slicing process has to know these characteristics in advance.

Deriving Slicing Patterns

The intention of a slicing pattern in the current context is to identify dynamically generated password fields. As previously discussed, we have elaborated a generally valid strategy that can be implemented in CryptoSlice to detect relevant input fields. Specifically, it works by backtracking particular method invocations and to verify whether they reflect a password field. Tracking the value itself, however, appears to be a challenging endeavour due to implicitly triggered control flows in combination with a multitude of possible methods to access the password. The design of an integrated slicing pattern would statically focus on a specific access method. For the current objective, however, a context-sensitive solution appears to be more constructive.

5.4 Static Slicing

Already introduced from an external perspective in Section 4.5.2, CryptoSlice implements a Slicing functionality that enables program statements to be followed in forward and backward direction. Starting from a given slicing criterion, the process determines the information flow between inter-connected code segments in Android applications and extracts the corresponding parts into a slice graph (see Section 5.5). In this section, we point out the challenges that have to be handled when performing static slicing on Smali code. After presenting the general concept, we focus on selected peculiarities and explain the implemented strategy.

5.4.1 General workflow

After deriving one or multiple slicing criteria from a given pattern, the slicing process starts by adding them to the internal *to-do list*. When requested by the slicer, it delivers the next object to track, including the register name and the location of the corresponding instruction. On the basis of this approach we are effectively able to control the slicing process and prevent the repeated analysis of already investigated data flows. Every tracked instructions causes a new register set to be added to the queue. The advantage of this approach is that we do not have to keep a state while tracking the individual instructions. Moreover, it facilitates adding further registers to the queue if the slicing process determines that other parameters of the currently tracked instruction also belong to the slice. A negligible drawback is the fact that the queue provides no look-ahead procedure. Practically, this would enable us to precisely choose the register to evaluate next. Hence, we could prefer tracking register sets with a lower fuzzy level as they indicate a higher accuracy in accordance with the slicing criterion.

Having obtained a particular register and instruction reference from the queue, the corresponding program statement is looked up. Subsequently, the instruction is decomposed into its individual components and the opcode is reduced to its base operation, e.g. `aput-boolean` to `aput`. As depicted in Figure 5.2, the base opcode is then assigned the corresponding group which is individually implemented for forward and backward slicing. Since Dalvik/Smali implements many identical base opcodes with different type suffixes, such as `-boolean`, `-byte`, and `-string`, we are able to combine them in common processing routines. In practice, this means that we do not have to implement a proprietary handling mechanism for every single type of bytecode.

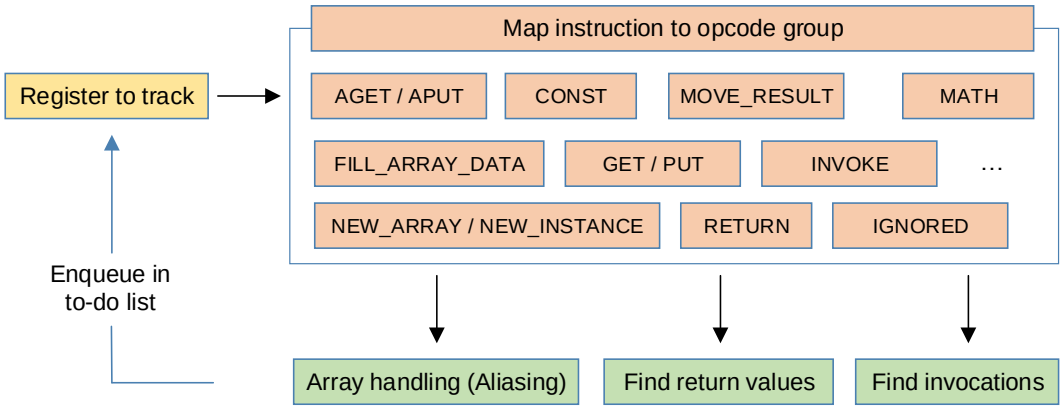

Figure 5.2: Static Slicing workflow

When working with opcode groups, the slicing process is likely to encounter read and write access to arrays, return calls with result values being passed back a caller method, and invocations of subsequent methods. Although the appropriate handling strategy differs for forward and backward slicing, our queue is suited for both mechanisms.

Array Handling (Aliasing)

Aside from plain values, registers may also hold references to arrays or objects, such as strings. For read access on arrays, Dalvik implements the opcode `aget`, while write operations are executed via `aput` instructions. In case a particular value is tracked in either forward or backward direction, it may be inserted into an array. Thereby, the register referencing the entire array is added to the slice and todo-list. This operation also elevates the overall slicing imprecision since any subsequent array read and write operation is now also tracked, disregarding the fact that not all values of the array are equally relevant with respect to the slicing criterion.

A particular situation occurs if multiple registers point to the same object. As exemplified in Listing 5.4, this happens if the reference to an array is copied into another register, using the `move-object` instruction. At this point, both registers refer to the same object state internally. Assuming that an `aput` instruction overwrites an array entry, the change implicitly also affects the other register. This problem is commonly known as *aliasing*. In the given sample code, the reference to `array2` in register `v2` is reset to `array1` by the `move-object` instruction. The final `aput` opcode implicitly causes an update of the value at `array1[0]`.

```
1  const/4 v0, 0x4
2  new-array v1, v0, [I                    # int[] array1 = new int[4];
3  const/4 v0, 0x3
4  new-array v2, v0, [I                    # int[] array2 = new int[3];
5
6  const/4 v0, 0x0
7  const/4 v5, 0x5
8  aput v5, v1, v0                         # array1[0] = 5;
9  move-object v2, v1                      # array2 = array1;
10 const/4 v6, 0x6
11 aput v6, v2, v0                         # array2[0] = 6;
```

Listing 5.4: Array Aliasing example

In order to cope with this problem, the slicing process has to be aware of object references in registers. Basically, there are two possibilities to distinguish them from ordinary values. First, if the register is defined by a `new-array` instruction and second, if a `new-instance` opcode creates a new object instance. During the copy operation, a register inherits the state of a source. The challenge for the slicing process, however, is to maintain a mapping of references with their associated registers. Since arrays are assigned an individual opcode for read and write operations, a change of the internal state can easily be recognized. For instances of externally defined objects, such as strings or lists, a value change is not clearly visible due to proprietary methods of the specific data structure that are typically unaccessible within the considered analysis scope.

Findings Invocations

In our slicing approach we abstain from computing a control flow graph over all methods before initiating the tracking process. As a consequence, we have to determine continuative invocations

on-line while slicing registers in forward or backward direction. Practically, this means that methods, called via `invoke` opcodes, are looked up in all available classes. Since all Smali files have already been parsed into an object-oriented representation, this instant search causes no I/O related overhead. In case a searched method signature cannot be found in the indicated class, matching declarations are looked up in all available superclasses. If a class cannot be found in the disassembled sources of an application, an external definition is assumed. Precisely, our slicing process does not consider classes that are induced by the Android framework or Apache Harmony. Since this unavailability may affect the accuracy of the resulting slices, we consider the return values of the according invocations as fuzzy (see Section 5.1.1).

5.4.2 Backward Slicing

The tracking process in backward direction traverses all preceding code statements that have influenced a specific register at the initial slicing criterion. Starting at a particular statement, CryptoSlice follows all usages of a register back to the point where it is defined. This approach is commonly referred to as *Use-Definition* (or short use-def). The resulting slice models the data flows of all variables that affect the starting point.

Example

In the following, we apply our slicing technique in backward direction on the Smali code provided in Listing 5.6. For a better understanding, Listing 5.5 contains the pendant in Java. The sample code shows the definition of a constant encryption key to be used with AES in the private method `deriveKey(...)`. The public method `getKeyBytes()` accesses the returned `SecretKeySpec` object and extracts the encryption key as byte array. For the backtracking process, we assume the slicing criterion $C = (13, \{v1\})$.

Initially, the code statement at line 13 is added to the slice. Subsequently, all preceding statements are traced, that have an influence on the register value of `v1`. In line 11, the process encounters a `move-result-object` instruction which denotes that the register holds the return value of a call to the method `getEncoded()`. Based on the return value type which is indicated that the end of the invocation statement, we see that the data type of `v1` is a byte array `[B`. Since resolving the called method is not feasible as it points to a Java library method, we continue with register `v0` since it causes `v1` to be declared after the call.

Another `invoke` statement follows in line 6, triggering a call to `deriveKey(...)`. The method is passed a string parameter, obviously held in register `v1`. However, the parameter is not yet added to the todo-list as it is still unclear whether it affects the result of the method call. Since the invoked method is found in the current analysis scope, all available `return` opcodes are looked up in the `deriveKey(...)` method and added to the todo-list. The slicing queue subsequently indicates the next register to track as $C = (37, \{v0\})$. Continuing the backward search, line 34 shows to involve the tracked register `v0` but apparently does not modify the object reference. Nevertheless, we see that the parameters in `v1` and `v2` presume to influence the object. Of course, since we cannot trace into the constructor of the `SecretKeySpec` object, this circumstance cannot be verified. Subsequently, we also add these registers to the todo-list but elevate their fuzzy level in order to annotate their uncertain relevance with regard to the tracked object. In total, this means that all three registers are now in the todo-list to be backtracked.

The already described procedure of `invoke` and `move-result-object` opcodes continues for the register `v1`. Since neither the `String`, nor the `StringBuilder` class can be accessed, each of these invocations raises the fuzzy level for the tracked register. The backtracking process for the

```
1  public class Test {
2    private static final String constPw = "1234567890abcdef";
3
4    private SecretKeySpec deriveKey(String password) {
5      return new SecretKeySpec(password.getBytes(), "AES");
6    }
7
8    public byte[] getKeyBytes() {
9      return deriveKey(constPwPart).getEncoded();
10   }
11 }
```

Listing 5.5: Backward slicing example (Java code)

```
1  .class public Lcrypto/Test;
2  .field private static final constPw:Ljava/lang/String; = "1234567890abcdef"
3
4  .method public getKeyBytes()[B
5    sget-object v1, Lcrypto/Test;->constPw:Ljava/lang/String;
6    invoke-virtual {p0, v1}, Lcrypto/Test;->deriveKey(Ljava/lang/String;)
7      Ljavax/crypto/spec/SecretKeySpec;
8    move-result-object v0
9
10   invoke-virtual {v0}, Ljavax/crypto/spec/SecretKeySpec;->getEncoded()[B
11   move-result-object v1
12
13   return-object v1
14 .end method
15
16 .method private deriveKey(Ljava/lang/String;)
17     Ljavax/crypto/spec/SecretKeySpec;
18   new-instance v0, Ljavax/crypto/spec/SecretKeySpec;
19   new-instance v1, Ljava/lang/StringBuilder;
20
21   invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V
22   invoke-virtual {v1, p1}, Ljava/lang/StringBuilder;->
23     append(Ljava/lang/String;)Ljava/lang/StringBuilder;
24   move-result-object v1
25
26   invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString()
27     Ljava/lang/String;
28   move-result-object v1
29
30   invoke-virtual {v1}, Ljava/lang/String;->getBytes()[B
31   move-result-object v1
32
33   const-string v2, "AES"
34   invoke-direct {v0, v1, v2}, Ljavax/crypto/spec/SecretKeySpec;-><init>
35     ([BLjava/lang/String;)V
36
37   return-object v0
38 .end method
```

Listing 5.6: Backward slicing example (Smali code)

registers `v0` and `v1` finally terminates at the corresponding `new-instance` opcodes in line 18 and 19. During the process, the register `p1` occurred at line 22 as influencing `v1`. When tracking this parameter register in backward direction, the process will not find a definition within the scope of the currently tracked method. Consequently, the context switches back to the formerly followed `getKeyBytes()` method. The register `p1` is substituted with the corresponding pendant `v1` at line 6 and finally resolves to a statically defined string, accessed by its field name `constPw`.

All registers which have been added to the todo-list during the slicing operation have been followed backwards until they were defined. This indicates to the slicing process that any theoretical predecessor statement would be unfeasible to affect the originally tracked slicing criterion.

5.4.3 Forward Slicing

In contrast to backtracking, forward slicing intends to find all parts which are influenced by the register at the slicing criterion. This implies that a forward slice contains all statements which depend on a value computed at the slicing criterion. Starting at the definition of a particular register, our framework traces all usages of the corresponding register in forward direction. This process is known as *Definition-Use* (or short def-use). The slicing process ends if all tracked registers resolve to constants or the fuzzy level exceeds a predefined threshold.

Example

The following example illustrates the slicing process in forward direction using the sample code in Listing 5.8. The Java code provided in Listing 5.7 points out the same functionality in an easier readable form. Our objective is to track the usage of the password field `etSecret` in forward direction. First declared in the method `onCreate(...)`, the value of the password field is accessed in the method `onClickGeneratePassword`. As a starting point for the tracking process, we assume the slicing criterion $C = (5, \{v0\})$ which refers to the password field, identified as `R.id.etSecret1`. During the compilation process, this reference has been replaced with a unique integer constant (see Section 5.3).

The first usage of the initial slicing criterion is immediately found in the first `invoke` instruction at line 6. Since the register affects the return value of the invocation, the register `v0` at line 7 is tracked further. The statements at lines 9-12 are at that point unrelated with the currently followed register. At line 15, the field variable `etSecret` is assigned the value of register `v0`. This implies that all subsequent read access operation (using `iget` instructions) to this field have to be tracked and are added to the todo-list. In our example, this affects the call at line 21. In the last `invoke` statement of the method `onCreate`, the reference contained in the tracked register `v0` is modified. However, this has no effect on the slice.

The tracking process continues at line 21 with register `v1`. The register is obviously employed to invoke the method `getText()`. As this method is not available to the current analysis scope, the fuzzy level is raised and set accordingly to the result at line 23. This procedure is similarly repeated for the invocations at the lines 24 and 29. Finally, at line 33 we encounter the last usage of `v1` and terminate the slicing process.

As a result, the slice contains all data flows between the slicing criterion and the endpoints at line 16 and 33. A subsequent analysis of this information flow could provide relevant details about the flow of a password. In this example, we have manually determined the trace of a password field to a `SecretKeySpec` object, holding an encryption key for data encipherment with AES.

```
1  public class Test extends Activity {
2    private EditText etSecret;
3
4    @Override
5    protected void onCreate(Bundle savedInstanceState) {
6      etSecret = (EditText) findViewById(R.id.etSecret1);
7      Typeface typeface = Typeface.createFromFile("Arial.ttf");
8      etSecret.setTypeface(typeface);
9    }
10
11   @Override
12   public void onClickGeneratePassword(View view) {
13     new SecretKeySpec(etSecret.getText().toString().getBytes(), "AES");
14   }
15 }
```

Listing 5.7: Forward slicing example (Java code)

```
1  .class public Lcrypto/Test;
2  .field private etSecret:Landroid/widget/EditText;
3
4  .method protected onCreate(Landroid/os/Bundle;)V
5    const v0, 0x7f0a0003
6    invoke-virtual {p0, v0}, Lcrypto/Test;->findViewById(I)Landroid/view/View;
7    move-result-object v0
8
9    const-string v1, "Arial.ttf"
10   invoke-static {v2, v1}, Landroid/graphics/Typeface;->
11     createFromFile(Ljava/lang/String;)Landroid/graphics/Typeface;
12   move-result-object v2
13
14   check-cast v0, Landroid/widget/EditText;
15   iput-object v0, p0, Lcrypto/Test;->etSecret:Landroid/widget/EditText;
16   invoke-virtual {v0, v2}, Landroid/widget/EditText;->
17     setTypeface(Landroid/graphics/Typeface;)V
18 .end method
19
20 .method public onClickGeneratePassword(Landroid/view/View;)V
21   iget-object v1, p0, Lcrypto/Test;->etSecret:Landroid/widget/EditText;
22   invoke-virtual {v1}, Landroid/widget/EditText;->getText()L
       android/text/Editable;
23   move-result-object v1
24
25   invoke-interface {v1}, Landroid/text/Editable;->toString()
26     Ljava/lang/String;
27   move-result-object v1
28
29   invoke-virtual {v1}, Ljava/lang/String;->getBytes()[B
30   move-result-object v1
31
32   const-string v2, "AES"
33   invoke-direct {v0, v1, v2}, Ljavax/crypto/spec/SecretKeySpec;-><init>
34     ([BLjava/lang/String;)V
35   return-void
36 .end method
```

Listing 5.8: Forward slicing example (Smali code)

5.5 Slicing graph

Forward and backward slicing are conceptually separated components that process the input from the to-do list and output slicing results to a dynamically built tree. Initially, the slicing criterion is set as root node, followed by all code statements that are contained in the slice. The generated graph is suited further analysis and evaluation purposes. For example, the security rules implemented for this thesis, dissect graphs and investigate individual execution paths (see Chapter 6).

The initial idea consisted in visualizing all data flows in one graph per slicing pattern. Due to the fact that a pattern can lead to multiple slicing criteria, this approach would cause incoherent flows of various criteria to be collated into a single representation. Aside from impeding the meaningfulness of the resulting graph, it would also lead to inconsistent results since overlapping data flows might occur multiple times. As a remedy, one graph is generated per slicing criterion. Compared with the output of an ordered list, the advantage of a directed graph representation is that it models connections between slicing nodes. Starting at a particular slicing criterion, it shows the information flow between the nodes and indicates the execution order.

Basically, program statements are inserted into the graph at the moment the slicer requests the next register from the todo-list. This operation, typically includes the names of the currently and previously tracked register, and a slicing node to establish a link with. This predecessor node is looked up in graph and referenced from within the newly added node. Priorly, it is checked whether the node to insert is already present in the graph. If this is the case, only a reference to a predecessor is added to the existing node, along with the name of the register in the current and previous statement. As a result, we manage to build a graph which contains all relevant program statements only. Nevertheless, it is capable of mapping all registers that are part of the slice and holds references to the according predecessors. Assuming that the slicing mechanism tracks loops, cycles are introduced into the graph. While this feature is suited to model repetitive invocations of methods, it has to be considered when traversing the graph. For this purpose, we employ a breadth-first search algorithm which is able to recognize already visited nodes. A similar approach is applied to extract individual executions paths from the graph.

Leaf nodes are added to the graph if the data flow has reached an endpoints or when slicing loses track. For instance, if registers are directed to unresolvable methods, an invariant endpoint is added since the data flow cannot be further evaluated. Leaf nodes are considered as constants. Aside from containing information about values that are assigned to registers, constants also explain why paths end at certain points. This is achieved by retaining metadata from slicing. For example, each constant is assigned a category which clearly defines the type of the underlying value. Similarly, in case tracking stops abruptly, constants are put in place to describe the cause. Using the available metadata, it can be highlighted whether slicing ended due to a native library call, an unknown API method, an event-triggered method (callback), or for other reasons.

After terminating a slicing operation, a graph requires post-processing in order to update the references between all slicing nodes. Assuming that a node B is linked from a node A and saves a reference to this node at the time of being inserted. In case node A is accessed in the further processing and adds another link to a predecessor, its internal reference changes. Now in order to avoid a dangling pointer, the JVM automatically creates a deep-copy at node B before node A is modified. While node B retains the anterior state this way, node A obtains a new reference. As a result, two states of node A exist whereas one potentially does not include all references to other nodes. The solution for this issue is to replace all, probably but not necessarily, outdated references with their current state. In other words, we discard outdated nodes and thereby, obtain a consistent data flow.

5.6 Limitations

Our static slicing implementation suffers from certain deficiencies that either raise the imprecision of data flows or in the worst case, fail to reproduce a coherent data flow. While minor issues may arise with specific applications, the accuracy of produced slices is also constrained by problems that occur independently of the investigated program. In the following, we briefly exemplify some known problems with our static slicing approach and discuss curative approaches.

Reference Tracking

Aside from holding explicit values, registers can also point to object references. Basically, redefining a tracked register ends an execution path since any subsequent register usage might no longer be related to the originally tracked content. However, in case a register holds a reference to an object or array and is reassigned the same object, our slicer loses track.

The sample code provided in Listing 5.9 illustrates the slicing operation that fails due the missing detection of references. Assuming that register `v8` is followed in forward direction, the register is passed to the method invoked at line 5. Since we are unable to track into this external method, we assume that the given parameter influences the object at register `v13`. The slicing process continues with the tracking of both registers. In the subsequent `iget` instruction, register `v13` is redefined with a reference to the object it already contains. As the slicing process is unable to detect this circumstance, it assumes that an unrelated value is assigned to the register.

```
1 const/4 v13, 0x0
2 const/4 v14, 0x0
3
4 iget-object v13, v0, LCrypto/Test;->hmac:Ljavax/crypto/Mac;
5 invoke-virtual {v13, v8}, Ljavax/crypto/Mac;->init(Ljava/security/Key;)V
6 iget-object v13, v0, LCrypto/Test;->hmac:Ljavax/crypto/Mac;
7 invoke-virtual {v13, v3}, Ljavax/crypto/Mac;->update([B)V
8 iget-object v13, v0, LCrypto/Test;->hmac:Ljavax/crypto/Mac;
9 invoke-virtual {v13, v4, v14}, Ljavax/crypto/Mac;->doFinal([BI)V
```

Listing 5.9: Limitation: Tracking object references

Implicit Control Flows

As already discussed in Section 5.3.2, Android supports the registration of event-triggered methods. They enable a predefined callback to be invoked whenever the event is signaled. The relation between event and callback is typically not obvious for the slicer and, hence, a consistent data flow cannot be determined. On Android, this problem is particularly predominant with UI components that are asynchronously triggered by user interaction.

As a remedy, it is conceivable to manually maintain a mapping of associations between events and callbacks. For example, assuming an invocation of the method `java.lang.Thread->start()` is determined by the slicing operation, it could automatically proceed with the internally called successor `java.lang.Thread->run()`. The drawback of this approach, however, is that a self-composed list would only cover a subset of all possible implicit control flow transitions. Nevertheless, the consideration of frequently occurring implicit information flows might significantly reduce imprecision.

External Methods

Related to the previously described problems is the usage of methods or classes that are not available within the considered analysis scope. Typically, this affects methods belonging to the Apache Harmony or Android framework. As exemplified in Section 5.1.1, a fuzzy level indicates uncertainty in analysis results, often caused by Java library methods.

By providing the slicer with libraries, specific to a particular Android version, the slicing process could follow most methods which otherwise are not trackable. Despite the gain of accuracy, slicing into system libraries would also dramatically raise the complexity and depth of the resulting graphs. Moreover, it is discussable how useful this contribution is for the overall slicing result.

As a compromise, it would feasible to create models of particular methods. For example, the data flow of `java.lang.System->arraycopy(src, srcPos, dest, destPos, length)` could be formally described and added to the slicing logic. In case the tracking process encounters this method, it could continue to follow `dest`, rather than losing track or elevating the fuzzy level.

Chapter 6

Security-critical Analysis Rules

This chapter presents the security rules that are implemented in CryptoSlice to detect security-related misconceptions in Android applications. They leverage the previously explained Slicing functionality and are tailored to identify and analyze common implementation flaws. For this thesis, the focus has mainly been put on the design of rules that reveal cryptography-related issues. Nevertheless, in order to illustrate the framework's simple adaptability for related application areas, we also include several rules that cover a broader scope of security-related code.

Prior to presenting the developed rules, Section 6.1 concisely confines our objectives and exemplifies problematic code that can hardly be uncovered by automated rules. The subsequent sections introduce the particular security rules. After a brief problem statement, we explain the chosen strategy for the detection of misuse and face correct implementation pendants. Depending on the rule, we also reflect on possible design shortcomings and the produced analysis output. Practically applied on a set of applications, Chapter 7 finally assesses the security rules' quality.

6.1 Objectives

Android supports the protection of sensitive data, such as entered passwords or personal information, by including the Java Cryptographic Architecture (JCA). Leveraging the provided high-level interfaces enables developers to access cryptography-related methods. Regardless of whether the actual implementation of a primitive is correct, it has to be supplied with input parameters that promote strong security. Due to the generic design of the JCA, developers are given the ability to configure integral cipher parameters deliberately. While this encourages great flexibility in a multitude of different deployment scenarios, the selection of weak parameters can drastically defeat the targeted level of security.

As a remedy, our framework includes a set of security rules that are intended to automatically perform a data flow analysis on parameters, used in security-critical code. The analysis of Smali code instead of Dalvik bytecode enables us to produce results that are readable and reproducible by software vendors. Using static slicing, we manage to track down wrongly chosen security attributes and to issue alerts with a conclusive problem statement.

The security rules make use of forward slicing to model the flow of sensitive information from the moment when user-entered data is made available to an application. The subsequent investigation of the computed paths can then reveal whether the input is subject to inadequate processing. For the analysis of parameters that are used with cryptography-related methods, data flows can be gathered using backward slicing.

Evidently, the elaborated rules are not capable of detecting all kind of weaknesses. Instead, they focus on the identification of common constructions and aim to distinguish proper usage from security-critical misconception. This mode of operation implies that inspected applications are susceptible to contain other issues which are not uncovered. The approach of countering misuse in further areas by implementing additional rules, is only feasible for operations that refer to standard APIs, such as the JCA. Regarding native code or custom implementations of cryptography, for example, our framework cannot take a final position.

6.2 No ECB Mode for Encryption

Our first rule states that Electronic Codebook (ECB) mode (see Section 2.7.2) must not be used for encryption. This paradigm emerges from the fact that, using the same encryption key, in ECB mode data blocks are enciphered individually from each other and cause identical message blocks to be transformed to identical ciphertext blocks. The independency of encrypted blocks also implies that the malicious substitution of a block has no impact on adjacent blocks. As a consequence, data patterns are not well hidden and message confidentiality may be compromised.

On Android, the `Cipher` API provides access to implementations of cryptographic schemes for the encryption and decryption of arbitrary data [1]. To request an instance of a particular cipher, an application has to invoke the method `getInstance`, passing a suitable *transformation* string as parameter. Typically, this value is composed of the desired algorithm name, a mode of operation, and the padding scheme to apply. For example, to request an object instance that provides AES in ECB mode with PKCS#5 padding, the transformation `AES/ECB/PKCS5Padding` has to be specified.

While it is indispensable to declare the algorithm to use, explicitly setting the mode and padding may be omitted. To fill the gap, the underlying Cryptographic Service Provider (CSP) relies on predefined values that do not necessarily reflect the recommended practice. Precisely, if the transformation indicates no operation mode, ECB mode is put in place. Moreover, the initially described problem with ECB is not limited to a specific cipher, such as AES but affects all symmetric block ciphers. Stream ciphers and asymmetric cryptosystems are not concerned since they do not involve an operation mode to repeatedly encipher blocks of contiguous data.

Detection Strategy

Based on the provided details about ECB mode, we can outline the workflow of this rule as follows:

1. Find all invocations of the method `javax.crypto.Cipher->getInstance(...)` and for each occurrence, backtrack the first parameter, holding the *transformation* value.

2. Using the slicing graph computed for a particular `Cipher` instance, identify all possible execution paths where the endpoint resembles a transformation or corresponds to a known symmetric block cipher, such as `AES`.

3. For each selected path, verify whether it includes a constant string that is part of a transformation, such as `/OFB/NoPadding`. If found, complete the algorithm name in the path endpoint with the determined mode and padding descriptor.

4. Analyze the final transformation value and raise an alert if the value either explicitly declares ECB mode, or specifies only the algorithm name.

[1] https://developer.android.com/reference/javax/crypto/Cipher.html

Backtracking the first argument of all overloaded `getInstance(...)` methods does not restrict the possible slicing result a priori. In other words, after the initial tracking step, we obtain a slicing graph and a list of appendant constants but cannot expect the trace to contain an algorithm descriptor. For example, if the value is heavily obfuscated or not even statically defined, a further analysis of the transformation value is not feasible.

Whether an execution trace leads to a descriptor at all is indicated by the path endpoint. By comparing it against a list of symmetric block ciphers, we exclude other algorithms that are not susceptible to the ECB mode problem. Due to the fact that the `Cipher` API can be used access algorithms in a variety of distinct CSPs, it is not sufficient to limit this list of known ciphers to only those cryptosystems that are natively supported on Android. Instead, we adhere to the repertoire of the Spongycastle [2] library, which implements the full version of BouncyCastle for Android and covers all relevant ciphers.

Having identified a matching slicing path where the endpoint represents solely an algorithm name, it is still not legitimate to assume that ECB mode is implicitly used. As depicted in Listing 6.1, it could occur that some part of the transformation depends on the actual execution context. Basically, the provided example contains four different transformations whereas the argument of the first `getInstance(...)` call leads to three possible execution paths. Inspection of the individual paths reveals that possible values are `AES`, `AES/CBC/PKCS5Padding`, and `AES/CTR/NoPadding`. The second invocation illustrates that the block cipher Twofish is preferred, in case the BouncyCastle library is registered. Taken from the source code of a real-world application, this example also highlights that the attainable security level and the question whether ECB mode is used, can be constrained by the availability of a supporting library.

```
1  String transformation = "AES";
2
3  int number = new Random().nextInt(10) % 4;
4  if (number == 0) {
5    transformation += "/CBC/PKCS5Padding";
6  } else if (number == 1) {
7    transformation += "/CTR/NoPadding";
8  }
9
10 Cipher c = Cipher.getInstance(transformation);
11 try {
12   c = Cipher.getInstance("Twofish/CBC/PKCS7Padding", "BC");
13 } catch (NoSuchAlgorithmException e) {}
```

Listing 6.1: Usage of ECB for encryption

6.3 No Non-random IV for CBC Encryption

Our second rule uncovers the usage of statically defined or predictable initialization vectors (IV) for encryption in CBC mode. For encipherment with feedback modes, such as CBC, an IV should ensure that data patterns are hidden and distinct ciphertexts are produced for the repeated encryption of identical plaintext blocks. Specifying a non-random IV leads to a deterministic and stateless encryption scheme that is susceptible to a chosen-plaintext attack (CPA). In this scenario, a malicious party can abuse the encryption scheme as an oracle (black box) to transform arbitrary

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

plaintext to ciphertext, without requiring the secret key. Assuming that the attacker learns the constant or predictable IV and XOR-combines it with a chosen plaintext, the encryption result will be deterministic.

```
1  // Constant IV
2  byte[] staticIv = new byte[] { 0x0f, 0x01, 0x02, 0x03, 0x04, 0x02, 0x01 };
3  IvParameterSpec ivParameterSpec = new IvParameterSpec(staticIv);
4
5  Cipher cipher1 = Cipher.getInstance("AES/CBC/PKCS7Padding");
6  cipher1.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
7
8  // Correct approach
9  Cipher cipher2 = Cipher.getInstance("AES/CBC/PKCS7Padding");
10 cipher2.init(Cipher.ENCRYPT_MODE, key);
11 byte[] randomIv = cipher2.getIV();
```

Listing 6.2: Constant and random IV for CBC encryption

After an instance of a particular transformation has been constructed using the `Cipher` API, it is possible to provide algorithm parameters by invoking the `init` method with the corresponding `AlgorithmParameterSpec` object. In order to process a particular IV as algorithm parameter, a byte array can be wrapped in an object of the type `IvParameterSpec` which is then passed to a cipher instance.

In the first part, Listing 6.2 illustrates the insecure practice of setting a constant byte array. The second part, in contrast, presents a more secure approach where the IV is randomly filled by the employed security provider upon each new encipherment process. For instance, the BouncyCastle library relies on the `SecureRandom` API to generate a cryptographically secure IV. Using the `Random` API instead, would undermine the overall security level and make the value predictable.

Detection Strategy

Our rule includes the following procedure for the identification of non-random IVs:

1. Find all invocations of those `javax.crypto.Cipher->init(...)` methods which include a `AlgorithmParameterSpec` object as second parameter and for each occurrence, backtrack this parameter.

2. Using the determined list of constants, verify whether an object of the type `javax.crypto.spec.IvParameterSpec` is created by calling its constructor. Abort, if none is found.

3. From each available slicing path, extract the subpath that begins at the `iv` argument, passed to the constructor of the `IvParameterSpec` object.

4. Investigate the subpath and raise an alert if the `iv` parameter is derived from a statically defined byte array, a string (e.g. using `String->getBytes()`), or a call to the cryptographically insecure `java.util.Random` API.

Based on the slicing results from the initial backtracking step, our first objective is to find out whether an object of the type `IvParameterSpec` is provided. If this condition is satisfied, we can conclude that the application manually defines an IV and thereby, overrides the default behaviour in which a strong IV is generated. Subsequently, we analyze the byte array which is

set in the constructor of the `IvParameterSpec` object. The according slice uncovers if the IV is composed of static values only or deduced from a constant, such as a string. Moreover, it reveals obviously predictable IVs, for example, when the `Random` API is employed. Aside from probing for specific indicators, we are naturally unable to make assumptions about the predictability of a value. Similarly, a non-static IV cannot not implicitly be considered unpredictable.

6.4 No Constant Encryption Keys

Keeping encryption keys secret is a vital requirement in order to prevent unrelated parties from accessing confidential data. Statically defined keys clearly violate this basic rule and render encryption useless. According to this paradigm, our security rule aims to detect hard-coded keys and raises an alert if a constant key is employed for symmetric encryption.

```
1 // Hard-coded key
2 SecretKey secretKey1 = new SecretKeySpec("secretkey".getBytes(), "AES");
3
4 // Better approach
5 KeyGenerator kg = KeyGenerator.getInstance("AES", "BC");
6 kg.init(256);
7 SecretKey secretKey2 = kg.generateKey();
```

Listing 6.3: Constant and randomly generated key for symmetric encryption

As demonstrated in Listing 6.3, a particular key can be declared by passing it as parameter to a `SecretKeySpec` constructor. If this byte array is derived from a constant value, it must not be used for symmetric encryption. Public-key cryptography, in comparison, operates with pairs of keys, whereas the key to encrypt a message is not a secret and can be made publicly available. Since the byte array of a public key can also be wrapped as a `SecretKeySpec` object, our security rule has to distinguish between keys for symmetric and asymmetric encryption algorithms. This can be achieved by inspecting the last constructor argument regarding the specified algorithm.

To obtain a secret key for encryption, the preferable approach consists in generating a random value with sufficient entropy. For this purpose, the JCA features a `KeyGenerator` which produces a secret key, according to a given algorithm and key size. The given example outlines the generation of a 256-bits key to be used for AES encryption.

Detection Strategy

CryptoSlice pursues the following strategy to reveal constant encryption keys that are used for symmetric encryption:

1. Find all invocations of `javax.crypto.SecretKeySpec->init(...)` methods and for each occurrence, backtrack the first parameter, holding the key.

2. Using the slicing graph computed for a particular `SecretKeySpec` instance, identify all contained constants and verify if the `key` parameter is derived from a statically defined byte array, or a string (e.g. using `String->getBytes()`).

3. If at least one possible key has been found, backtrack the last parameter of the corresponding `SecretKeySpec` instance and extract all feasible execution paths from the resulting graph.

4. Investigate each available slicing path and verify whether one of the following asymmetric encryption schemes is specified: `DHIES`, `ECIES`, `ElGamal`, `RSA`.

5. If the provided algorithm does not belong to a known public-key cryptosystem, we conclude that the statically defined key is used for symmetric encryption and raise an alert since the mandatory secrecy is not given.

Backtracking the `key` parameter of `SecretKeySpec` constructors does not not necessarily lead to an unambiguous result. In practice, it is probable that a statically defined value is concatenated with a non-constant input. Declaring the resulting key as a constant would not resemble the actual situation since a the non-constant part might still provide enough entropy. Of course, the contrary would be true if the statically defined part prevails. As a consequence, we decided for a compromise: if statically defined values influence the resulting key, a warning is raised although the key is not entirely constant. Thereby, we emphasize the fact that the key might be substantially affected, rather than stating that it is statically defined.

6.5 No Constant Passwords or Salts for PBE

Password-based encryption (PBE) misses the intended purpose if the induced password or salt value is statically defined. Basically, a cryptographically secure key is deduced from a given secret by repeating a key derivation function (KDF) multiple times. A randomly chosen salt value ensures that the derived key is unique and slows down brute-force and dictionary-based attacks dramatically. However, if the password is hard-coded, the derived encryption key has to be considered broken since the confidentiality of the encrypted data is no longer guaranteed. Likewise, specifying a constant salt value contradicts the goal of using PBE to hinder table-based attacks. Practically occurring in the same construction, our rule targets both misconceptions.

```
1  SecretKeyFactory fac = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
2
3  // Hard-coded password and salt
4  char[] password = "secretkey".toCharArray();
5  KeySpec keySpec1 = new PBEKeySpec(password, "salt".getBytes(), 20000, 256);
6  Key key1 = fac.generateSecret(keySpec1);
7
8  // Better approach
9  int keyLen = 256;
10 byte[] randomSalt = new byte[keyLen / 8];
11 new SecureRandom().nextBytes(randomSalt);
12
13 char[] password2 = <external source, e.g. password input field>
14 KeySpec keySpec2 = new PBEKeySpec(password2, randomSalt, 20000, keyLen);
15 Key key2 = fac.generateSecret(keySpec2);
```

Listing 6.4: PBKDF2 key derivation (JCA)

As specified in the PKCS#5 standard[3], the PBKDF2 key derivation function involves a password, salt value, iteration count, and the desired key length. On Android, these parameters can be declared using the `PBEKeySpec` API. Subsequently, a `SecretKeyFactory` instance transforms the password to an encryption key by invoking its `generateSecret` method.

[3] https://ietf.org/rfc/rfc2898.txt

The sample code in Listing 6.4 demonstrates two key derivations based on a given password. After outlining the problematic practice of using a constant password and salt value, the second part proposes a more secure alternative which relies on dynamically generated or induced values only. As annotated, the password could, for example, correspond to a user-provided input value which is dynamically set at run-time. The salt should originate from a high-quality random source, e.g. the `SecureRandom` API, rather than being defined as a constant. Typically, its size corresponds to the desired length of the key. Hence, assuming that a 256-bits key is derived for AES encryption, the byte array holding the salt would consist of 32 bytes with random data (256 / 8 = 32).

Detection Strategy

Using the following strategy, we are able to identify constant passwords or salts, used with PBE:

1. Backtrack relevant parameters from the corresponding methods:
   (a) Find all method invocations of `javax.crypto.spec.PBEKeySpec->init(...)` and for each occurrence, slice the first parameter, holding the password.
   (b) If a `PBEKeySpec` method signature involves multiple arguments, track the salt value provided by the second parameter.
   (c) Identify all calls to `javax.crypto.spec.PBEParameterSpec->init(...)` methods and compute the slice for the `salt` argument.
2. Investigate all computed slicing graphs and determine for each whether at least one execution path supplies the `password` or `salt` parameter with constant input values.
3. An alert is raised if the parameters are derived from a statically defined char or byte array, a string (e.g. using `String->toCharArray()` or `String->getBytes()`), or a call to the cryptographically insecure `java.util.Random` API.

Integrating the analysis of constant password and salt values into a single rule enables us to save redundant searches for the same method invocations. Unless the method signatures allow only for one argument, such as the `salt` parameter in the `PBEParameterSpec` constructor, we are able to derive multiple slicing criteria for the different parameters of a particular program statement.

By focussing only on JCA provided APIs, our rule does not only disregard all implementations of custom cryptography but also ignores calls to libraries which offer the same functionality apart from the provider hierarchy. On Android, the BouncyCastle library, for instance, externally exposes functionality for key derivation purposes. Android applications can directly access the corresponding methods and thereby, circumvent our detection. Listing 6.5 demonstrates the workflow of deriving a PBKDF2 key without the JCA. The example relies on a constant password and salt value and produces the same key as the problematic statement in the previous code sample. As a remedy, the present security rule could be extended to also cover other relevant method calls.

```
1 byte[] password = "secretkey".getBytes();
2
3 PBEParametersGenerator gen = new PKCS5S2ParametersGenerator();
4 gen.init(password, "salt".getBytes(), 20000);
5 KeyParameter keyParam = (KeyParameter)gen.generateDerivedParameters(256);
6 byte[] derivedKey = keyParam.getKey();
```

Listing 6.5: PBKDF2 key derivation (BouncyCastle library)

6.6 Not Fewer than 1000 Iterations for PBE

Using a low iteration count for Password-Based Encryption (PBE) significantly reduces the costs and computational complexity of table-based attacks on derived keys. Intended to raise the difficulty of attacks, it should be set as high as possible while keeping performance at a reasonable level. In 2010, a minimum of 1000 iterations has been advised [47]. Considering the emerging capabilities of mobile devices, higher values might be suitable without noticeable delay.

Typically, the iteration count for PBE is declared using the `PBEKeySpec` or `PBEParameterSpec` API. Along with the password, salt, and desired key length, the parameters are passed to a key derivation function, such as PBKDF2. The sample code in Listing 6.4 and 6.5 employs an iteration count of 20000 which offers a comparably large security margin for current applications. In accordance with the PKCS#5 standard, our rule targets constructions that use less than the minimally recommended 1000 iterations.

Detection Strategy

Similar to the strategy of the previous rule, we detect low iteration counts for PBE as follows:

1. Backtrack relevant parameters from the corresponding methods:
   (a) Find all method invocations of `javax.crypto.spec.PBEKeySpec->init(...)` and for each occurrence, slice the second parameter, holding the iteration count.
   (b) Identify all calls to `javax.crypto.spec.PBEParameterSpec->init(...)` methods and compute the slice for the `iterationCount` argument.
2. From each computed slicing graph, extract all feasible execution paths and thereof determine all endpoints that include a constant integer value.
3. Compare the determined integers with the predefined threshold of 1000 and raise an alert if a PBE construction uses less iterations.

The detection workflow starts by obtaining the slicing graphs for all JCA-related method calls. Thereafter, the focus is put on identifying integer values that describe an iteration count. Since the backtracking process ends with the definition of constants, we know that a slicing graph includes statically defined iteration counts as leaf nodes. Extracting them is possible by searching for all nodes that are not referencing any subsequent slice nodes. Alternatively, as described in the workflow, we split the graph into individual execution paths and analyze their endpoints.

Practically, it is possible that more than one iteration count is found for a particular PBE construction. For example, some applications derive the amount of iterations to use dynamically by benchmarking the underlying system. As a consequence, our rule would fail to detect the final iteration count since it is not provided as a constant. Evidently, this is a general drawback of static analysis, rather than a deficiency of our concept.

6.7 No Static Seeds for SecureRandom

Based on an initial seed value, pseudorandom number generators (PRNG) apply algorithms to derive a sequence of random bits. If a PRNG is seeded with a statically defined value, it will produce a deterministic output which is not suited for security-critical applications.

With Android 4.2, the default PRNG provider has been changed from Apache Harmony to the native AndroidOpenSSL. Prior to that, it was possible to override the internally designated seed with a custom value which, in case it was constant, caused the generation of deterministic output values. The provider transition ensured that user-provided seed values were appended, rather than substituting the entire internal state of the PRNG. Inherently, the change also resolved the issue with statically defined seeds. Nevertheless, the problem still exists with all devices running Android 4.1 or lower (currently 29%[4]). Moreover, it is still possible to manually specify a different provider in order to force the former behaviour.

```
1 // Problematic PRNG
2 SecureRandom rand1 = new SecureRandom("constant seed".getBytes());
3 rand1 = SecureRandom.getInstance("SHA1PRNG", "Crypto");
4 rand1.setSeed("constant seed".getBytes());
5 System.out.println(rand1.nextInt());
6
7 // Correct approach
8 SecureRandom rand2 = new SecureRandom();
9 System.out.println(rand2.nextInt());
```

Listing 6.6: Deriving random values using the SecureRandom API

By default, Android utilizes the algorithm `SHA1PRNG` when instances of the `SecureRandom` API are created. Alternatively, it is possible to select a particular PRNG by invoking the method `getInstance` with a suitable algorithm name. Unless explicitly specified, the value of the initial seed depends on the underlying provider and implementation. Typically, the output of the system PRNG `/dev/urandom` is taken since this source of entropy is considered unpredictable.

As demonstrated in Listing 6.6, a seed can be explicitly set in the `SecureRandom` constructor or declared by invoking the method `setSeed`. Despite the default provider change with Android 4.2, this practice remains clearly disadvised [5] but is no longer problematic. Of course, if the former Apache Harmony provider `Crypto` is manually enforced, the problem persists regardless of the employed Android version.

Detection Strategy

Our rule includes the following workflow to uncover constant seeds values:

1. Backtrack relevant parameters from the corresponding methods:
   (a) Find all method invocations of `java.security.SecureRandom->init(...)` and for each occurrence, slice the first parameter, holding the byte array with the seed.
   (b) Identify all calls to `java.security.SecureRandom->setSeed(...)` methods and compute the slice for the `seed` argument, which may consist of a byte array or eight bytes contained in a 64-bits integer (`long`).
2. Investigate all computed slicing graphs and determine for each whether at least one execution path supplies the `seed` parameter with constant input values.
3. An alert is raised if the parameter is derived from a statically defined byte array, a string (e.g. using `String->getBytes()`), or a 64-bits integer.

[4] https://developer.android.com/about/dashboards/
[5] https://developer.android.com/reference/java/security/SecureRandom.html

Aside from revealing whether a constant seed is set, the slice graphs for `SecureRandom->setSeed(...)` methods may also contain references to `SecureRandom->nextBytes(...)`. This indicates that the PRNG has been seeded anew after deriving a pseudorandom sequence. Depending on the implementation of the underlying PRNG, this may either cause the existing seed to be supplemented or replaced with the new value.

Our detection workflow is designed to focus solely on the methods that enable the setting of a seed value. By disregarding the used algorithm and provider, we are unable to state whether the Apache Harmony provider has been enforced. Practically, however, this circumstance plays a minor role since it does not affect the objective of our rule. In other words, we *always* consider the use of a statically defined seed problematic, irrespective of the employed JCA provider. Of course, the practical impact finally depends on the used Android version and provider implementation.

6.8 No MessageDigest Without Content

Calling a hash function without a message results in a deterministic and predictable digest. Typically, this problem arises unintentionally in methods with a complex data flow or when setting the message digest input depends on a condition. Typically, one-way hash functions are designed to make the output easily computable but impossibly to invert. A constantly empty input value, however, leads to a hash value that does not withstand trivial brute-force or dictionary-based attacks.

```
1  // Hash computation from empty input (digest1 and digest2)
2  MessageDigest md1 = MessageDigest.getInstance("SHA-256");
3  ... // no update(...) call
4  byte[] digest1 = md1.digest();
5
6  md1.update("content".getBytes());
7  md1.digest(); // resets the internal state
8  byte[] digest2 = md1.digest();
9
10 // Correct usage
11 MessageDigest md2 = MessageDigest.getInstance("SHA-256");
12 md2.update("content".getBytes());
13 byte[] digest3 = md2.digest();
14 byte[] digest4 = md2.digest("content");
```

Listing 6.7: MessageDigest example with and without content

Prior to deriving a message digest, an instance of a particular hash algorithm has to be constructed using the `MessageDigest` API. Therefore, the method `getInstance` has to be invoked with a corresponding algorithm. Subsequently, it is possible to set input data using the method `update(...)` The hash computation is completed with a call to a `digest(...)` method which finally delivers the digest. If the latter method is supplied with input data, it first performs the update step internally and then computes the hash value.

The actual problem arises if the `update` method is never called and `digest` is invoked without input data. The resulting digest, hence, is based on an empty input. After demonstrating the faulty workflow, Listing 6.7 points out secure alternatives. As can be seen, not the call to `digest()` without parameters is the issue but the absence of preceding update statements. Invoking only the terminating method with an input is also a legitimate option.

Detection Strategy

Using the following workflow, we are able to detect MessageDigest instances without content:

1. Find all invocations of `java.security.MessageDigest->getInstance(...)` methods and for each occurrence, track the instance object in forward direction.
2. Using the slicing graph computed for a particular `MessageDigest` instance, verify for each available execution path whether the method `java.security.MessageDigest->digest()` is present at least once. Do not proceed with a path if no invocation can be found.
3. From each matching execution path, extract the subpaths between the slicing criterion and one or multiple `digest()` invocations.
4. Investigate all extracted subpaths and check if an `update(...)` method is called before `digest()`. Likewise, ensure that the methods `reset()` and `digest()` are *not* invoked between `update(...)` and the trailing hash computation call.
5. If the presence of an `update(...)` statement could not yet be determined, verify if the path contains calls to `write(...)` methods of the classes `java.security.DigestOutputStream` or `java.security.DigestInputStream`. If found, imply the setting of an input value.
6. An alert is raised if the mandatory input value appears to be undeclared.

Tracking the instance object in forward direction enables us to capture all statements that affect a particular instance. A subsequent inspection of the resulting data flow discloses whether an input value is set using the `update(...)` method. Having ascertained that an input is set, the remaining statements until the final `digest()` method need to be analyzed regarding their effect on the overall input. For example, while updating a message digest instance with a `null` value would fail with a `NullPointerException`, the internal state can still be cleared by calling the `reset()` method. Likewise, it is conceivable that a `digest()` method is invoked immediately *after* another `digest(...)` call. As previously illustrated in Listing 6.7, the result would be a hash value that is derived from an empty internal state.

Finally, we need to take into account that input values can alternatively be set using auxiliary classes, such as `DigestInputStream` and `DigestOutputStream`. Since the slicing process is conducted in forward direction, it also comprises calls to possible `write(...)` methods that would equally indicate the setting of input values. By refining the security rule to consider these cases as well, we increase its accuracy and at the same time, prevent false positive alerts.

6.9 No Password Leaks

Evaluating the data flow of passwords regarding security aspects is challenging since the severity of problems may depend on the context of an application. For example, it might be inappropriate to flag an application insecure due to the fact that a password does not undergo a cryptographic transformation. Of course, the opposite can be true for applications where cryptography is inevitable in order to protect sensitive data.

Considering passwords as sensitive information, our security rule focusses on general misconceptions that substantially affect its secrecy. For instance, one paradigm states that passwords must not be written to a logfile. This emerges from the fact that the mandatory confidentiality is

no longer given as soon as an unintended party is able to learn secret credentials. By analyzing the data flow between a password field and one or multiple endpoints, we aim to answer the following questions:

- Is an entered password written to an output file?
- Is a password sent to a logging function and leaked to a logfile?
- Does a password undergo a cryptographic transformation?

If one of the first two conditions is satisfied, the security of an entered password is clearly impaired. The latter question specifically depends on the investigated application. For example, under normal circumstances there is no need for a Mobile Banking application to transform a password in order to login to the service behind. In contrast, a program which intends to securely store data protected by a user password, undoubtedly should apply cryptography for key derivation and data encipherment.

Detection Strategy

Using the following strategy, we intend to evaluate the questions listed above:

1. Identify available password fields by applying the patterns, elaborated in Listing 5.2, and for each occurrence, track all resource usages in forward direction.
2. From each computed slicing graph, extract all feasible execution paths and thereon evaluate the following conditions:
   (a) Raise an alert if the data flow includes calls to `write(...)` methods of the classes or any known subclasses of `java.io.OutputStream` and `java.io.FileWriter`. Also detect when passwords are printed using the class `java.io.PrintWriter`.
   (b) Check if a password is sent to log output or leaked to a logfile using methods of the `android.util.Log` API. Issue a warning if corresponding calls have been found.
   (c) Verify if a password is processed by security-related APIs, exposed in the packages `java.security.*` and `javax.crypto.*`. If found, emit a notification.

The detection workflow starts by obtaining the slicing graphs for all password fields. Initially containing the offset of the password resource (see Section 5.3.1), the data flow of an execution path models all program statements that are affected by the input field. Inspecting the graph enables us to search specific accessors that are known to implement the questioned behaviour.

Having confirmed the presence of a particular class or method, the remaining task consists in ensuring that the found method or class sufficiently relates to the original password field. In other words, we have to measure the uncertainty that a particular access method actually works with the password field, rather than any unrelated object. This information is provided by the *fuzzy level* (see Section 5.1.1) which expresses the likelihood that the value of the register at a particular execution statement still equals the value of the initially tracked register.

Whether the analysis targets classes, rather than specific methods has been decided depending on the individual functionality. For example, since the logging API of Android exposes only methods that inevitably log a given input, the entire class is blacklisted. Subsequently, we can abstain from measuring the fuzzy level of the register that is passed to a concrete method call. Instead, we assess the accordance at the highest possible level, such as a class or package.

Chapter 7

Evaluation

This chapter provides the evaluation results of the security rules that have been presented in the preceding chapter. Leveraging the static slicing functionality of our CryptoSlice framework, the rules are tailored to detect and analyze common implementation flaws. By applying them manually and automated on a set of applications that process sensitive data, we evaluate their accuracy and gain an insight into the distribution of security-related misconceptions in Android applications.

Prior to presenting our results, Section 7.1 first introduces the applied evaluation methodology and briefly describes the tested dataset. The subsequent section 7.2 exposes and discusses the findings of each particular security rule. Depending on the rule, we also reflect on possible design shortcomings and propose potential ameliorations.

7.1 Evaluation Process

The evaluation of the provided security rules is conducted both automated and manually on the same dataset. While the first process is capable of pointing out the prevalence of security-critical weaknesses in Android applications, the manual analysis is suited to assess the quality and accuracy of the implemented rules. Evidently, the scope of the evaluation does not extend to the discovery of *all* security problems in the investigated applications. Instead, both approaches focus solely on the aspects and misconceptions that are also targeted by our rules.

For the *automated evaluation*, we use CryptoSlice to apply the security rules on a set of previously selected applications. During the analysis process, the framework automatically adds all findings and conclusions to an XML report that is created individually for each investigated program. Based on the emitted reports, we are able to perform a conclusive statistical study which is subsequently presented in this chapter.

In the *manual evaluation* scenario, we assess the suitability of the implemented patterns by comparing the results from the automatically conducted analysis with manual findings in disassembled Smali code (see Section 2.5). Consequently, it is possible to draw a series of conclusions: By determining whether a security rule properly matches the constructions it is designed for enables us to detect possible deficiencies in our concept. Likewise, it is probable to locate obvious false positives that result from an imprecise slicing operation. Moreover, manual analysis might reveal conditions that cannot be handled by our rules so far. Finally, we are able to learn whether a detected misconception does indeed correspond to an actual security-critical problem or flawed security property.

The subsequent evaluation conceptually combines both approaches into a single representation while pointing out notable findings, irregularities, and peculiar analysis results.

7.1.1 Investigated Dataset

For the current evaluation scenario, we have downloaded 253 Android applications from the official Google Play store [1]. The dataset has been compiled between January and April 2014 and includes only applications with more than 10.000 installations, as indicated by the marketplace.

#	Category	Description
100	Password Manager	Applications of this category are intended to securely store and manage user credentials in an encrypted form. Typically, users authenticate to the manager with a master password. Using our security rules, we aim to assess whether these applications implement the stipulated encryption adequately.
55	Mobile Banking	Mobile Banking applications provide customers with a well-adapted interface to banking services. A login usually involves personal bank account data, including a secret password. Thus, a particular focus has to be put on the security of the entered data.
51	Cloud Storage	Multiple cloud service provider support data privacy by encrypting files on the device before they are sent to the service. We want to assess the quality of the promised encryption and determine the amount of applications offering data protection.
38	Messenger	Many messengers claim to securely exchange personal conversation. Using our security rules, we can verify whether they rely on security-related APIs in order to achieve this.
9	Secure Container	Container applications safeguard personal information, such as emails and documents in segregated data stores. In order to prevent unauthorized access to this data, security mechanisms have to be correctly employed.

Table 7.1: Description of the analyzed dataset (253 applications)

The listing in Table 7.1 outlines the exact composition of our dataset by assigning the applications to the corresponding category. As can be seen, we put an emphasis on the evaluation of password manager applications since they inevitably have to apply cryptography in order to securely process user-entered passwords. Similarly, all remaining applications that work with sensitive user data are supposed to invoke protective methods.

Basically, the applications from our dataset have been chosen empirically with respect to the common security requirements of the according category. An automatically performed assessment on the usage of security-related APIs has revealed that 18 of the selected 253 applications do *not* invoke methods from the packages `java.security.*` and `javax.crypto.*`. Consequently, we assume that the concerned applications either abstain from implementing security methods or rely on different approaches. This circumstance, however, is beyond the scope of our analysis.

[1] https://play.google.com

7.2 Security-critical Analysis Rules

In total, we have applied our security rules on 253 pre-selected Android applications. Beyond the investigated programs, 5 could not be analyzed automatically by our framework since the slicing process was either aborted after the defined threshold of 25 minutes or surpassed the limit of 80.000 tracked registers. The analysis of another set of three applications failed due to limitations in the amount of usable memory. Precisely, during the pre-processing step, the automated analysis ran out of memory while parsing the Smali code into an object-oriented representation. A manual review of the affected programs revealed that their Dalvik bytecode is heavily obfuscated using ProGuard and related mechanisms (see Section 2.5.1).

During the analysis, we found that many applications include the same third-party libraries for advertisements, statistics, and other functionality. Assuming that these libraries also include security-related code in libraries, such as BouncyCastle or SpongyCastle, we prevent a possible bias in our analysis results by automatically filtering known libraries based on a blacklisting approach. Rather than employing heuristics to detect them, we presume that applications do not intentionally conceal the use of third-party libraries. Hence, CryptoSlice relies on a list of package names to detect and exclude unwanted Smali classes right before transforming them into objects. As a side effect of this approach, we prevent the waste of memory and ensure that non-relevant files are already omitted from being loaded at an early stage.

The automated evaluation of all security rules showed that out of 245 analyzable applications, 10 did not contain security-critical constructions that were targeted by our patterns. In other words, except for these ten, all investigated applications contained JCA-provided API calls or trackable password fields. A manual verification confirmed that the concerned programs either contain large portions of Android native code or are based on cross-platform development frameworks, such as Cordova [2] or Appcelerator [3]. As previously stated, our analysis purely targets the use of security-related APIs in application code, thus factoring out enclosed usage in third-party libraries.

In the following, we present the evaluation results for the remaining 235 inspected applications whereas each implements at least one targeted API invocation or contains a trackable password field. For every rule, we explain the statistical findings, discuss the quality of the tracked patterns and outline the security-critical impact for affected applications.

7.2.1 No ECB Mode for Encryption

The first rule states that Electronic Codebook (ECB) mode must not be used for encryption. 95 out of 253 inspected applications do not coincide with this paradigm and use ECB mode.

In total, we determined that 162 applications include 543 calls to the `getInstance(...)` methods of the `javax.crypto.Cipher` API. This means that on average, each application invokes the methods more than three times with a particular transformation string. Considering that the `Cipher` API creates a separate object instance for the encryption and decryption of arbitrary data, it is reasonable that the construction practically occurs multiple times within a single application.

By inspecting the 543 found constructions, we were able to successfully identify 566 algorithm descriptors. Another 33 cipher transformation strings could not be determined. Having found more algorithm names than method calls indicates that certain applications envisage multiple algorithms for a particular cipher instance. In that case, it is conceivable that the chosen transformation

[2] https://cordova.apache.org
[3] https://www.appcelerator.com

depends on the actual execution context. Section 6.2 provides a code sample which illustrates an approach on how to consider this case in practice.

In the manual application review and study of the available data flows we attempted to find out the reason for 33 undetected algorithm specifiers. The investigation revealed that the descriptors for 18 of these ciphers could not be determined due to the employed obfuscation of strings. Since our rule compares found algorithms against a list of known ciphers, another possibility would have been that we simply did not recognize the indicated names. However, this assumption has also not been confirmed for the remaining 15 ciphers which were not be detected due to invocations from native code or an incomplete data flow.

Matching the 566 found algorithm specifiers against the list of known ciphers with ECB mode discloses that 43% or 242 ciphers include ECB mode, while 57% or 324 occurences rely on different modes of operation. As can be observed in Figure 7.1, the vast majority of 76% or 185 symmetric ciphers are applied without declaring the mode and padding to use. Consequently, the underlying Cryptographic Service Provider (CSP) implicitly puts ECB mode in place. The remaining 24% or 57 algorithms explicitly instantiate a symmetric block cipher with ECB mode. It is worth mentioning that the presented numbers do not include occurrences of stream ciphers (e.g. `ARCFOUR/ECB/NoPadding`), or asymmetric cryptosystems (e.g. `RSA/ECB/NoPadding`) since they internally do not involve an operation mode to repeatedly encipher blocks of contiguous data.

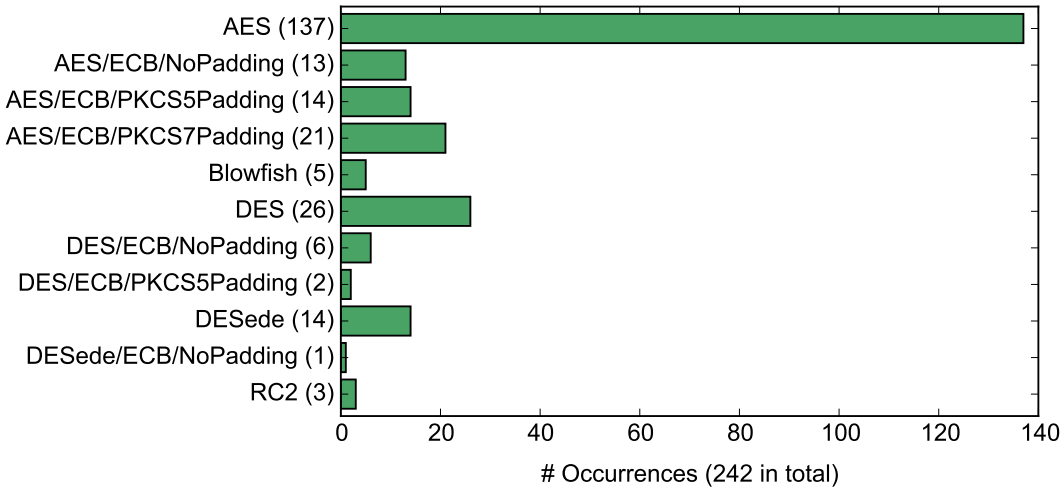

Figure 7.1: Distribution of ECB mode ciphers for symmetric encryption

The significant prevalence of AES without an explicitly set mode and padding leads to the assumption that many developers are not aware of the associated implications. Due to the fact that commonly used CSPs on Android, such as BouncyCastle or AndroidOpenSSL, configure ciphers to work with ECB mode by default, adequate security properties are not enforced by these third-party entities. Also notable is the usage of the broken DES and RC2 ciphers. Although both algorithms are considered insecure for current applications, they are employed by 16% or 29 cipher instances. The manual review confirmed that these algorithms are indeed practically applied and do not belong to a cryptographic library.

Figure 7.2 depicts the general distribution of transformation strings used with the Cipher API. Less frequently employed algorithm descriptors of the BouncyCastle library (`PBEWith<digest>And<encryption>`) and remaining ciphers are subsumed unter the item `Other`. As can be seen, with a usage rate of 19.3% `AES/CBC/PKCS5Padding` is the second most adopted algorithm after

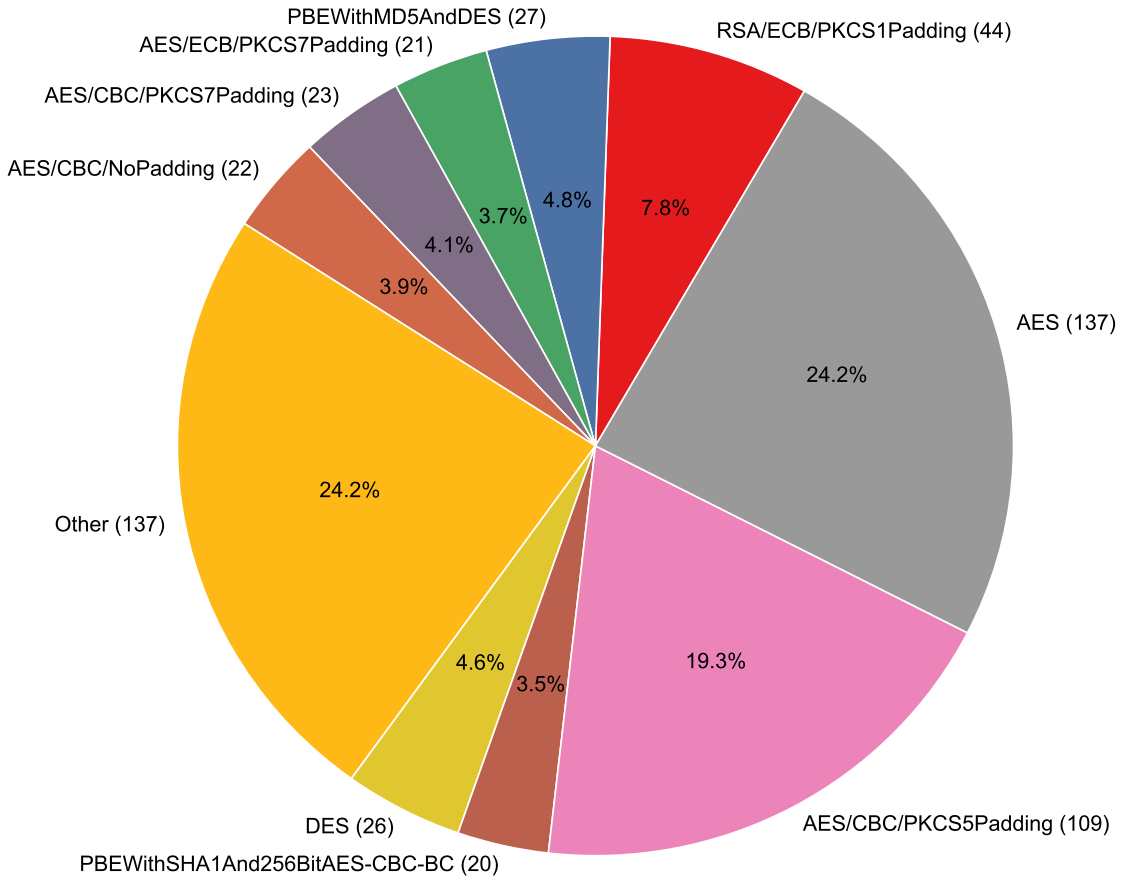

Figure 7.2: Usage distribution of algorithms used with `Cipher->getInstance(...)` (566 occurrences in total)

plain AES with 24.2%. Equivalent to the variant with PKCS#7 padding, this transformation represents the preferable replacement for AES with ECB mode, if setup with a random initialization vector (IV). With a rate of 7.8%, the only exposed asymmetric encryption scheme is RSA. The predominance of this public-key algorithm is not surprising since current Android versions do not natively include support for alternatives like DHIES or ECIES.

While presuming that ECB mode must not be employed for encryption in general, our security rule disregards situations where the use of this operation mode would not raise severe security concerns. This is the case, for example, when the message to encrypt does not exceed the underlying block size, or if all data blocks are unique. Since static analysis is unfeasible to make assumptions about the used plaintext value to encrypt, we are unable to take particular situations into account where ECB mode might not be problematic. Moreover, the manual analysis of the concerned applications unveiled that no application which limits to default security properties, actually intended to perform encryption on random data or values that definitely do not surpass one message block.

7.2.2 No Non-random IV for CBC Encryption

The second rule targets the usage of statically defined or predictable initialization vectors (IV) for encryption with CBC mode. Performed on a set of 253 applications, our evaluation identified 28 programs which specify a constant IV. The underlying cipher constructions are, thus, vulnerable against a chosen-plaintext attack (CPA) where an attacker can abuse the encryption scheme as an oracle to transform arbitrary plaintext to ciphertext.

We disclosed that 97 inspected applications include 264 invocations of the `init(...)` methods of the `javax.crypto.Cipher` API in order to manually define parameters to use with the underlying cryptosystem. Backtracking all passed objects of type `AlgorithmParameterSpec` showed that 215 instances reference manual specifications of initialization vectors as `IvParameterSpec` objects. In contrast, 49 instances specified different algorithm parameters, e.g. for password-based encryption (`PBEParameterSpec`) or Elliptic Curve Cryptography (`ECParameterSpec`).

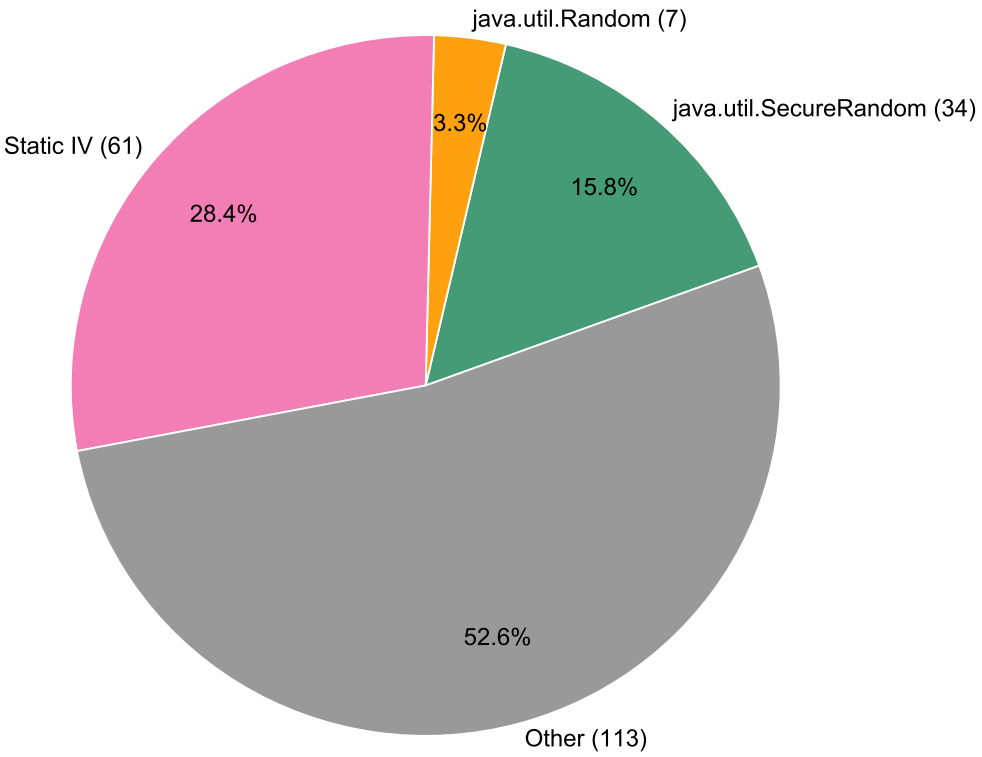

Figure 7.3: Distribution of initialization vectors used with `Cipher->init(...)` (215 occurrences)

Focussing solely on descriptors of initialization vectors, our evaluation uncovered that a constant IV could be found in 28.4% or 61 cases. The manual verification of these results confirmed the accuracy of these findings and revealed that in 42 cases the IV is derived from a static string value using `String->getBytes()`. The 19 remaining constants are built upon hard-coded byte arrays with varying values. As can be seen in Figure 7.3, 36% or 41 IV values are not static but leverage the `Random` or `SecureRandom` API. While the latter class with 34 invocations is legiti-

mate to generate a cryptographically strong value, the use of the first mentioned API in 7 cases undermines the overall security level and makes the IV predictable. It is worth mentioning that aside from verifying the usage of the `Random` API, we are naturally unable to make assumptions about the predictability of IVs. Assuming, for instance, that an application relies on user input as a basis for its cryptographic parameters, we can only declare that the value is not composed of static values or deduced from a constant, such as a string.

The large remainder of 52.6% or 113 invocations includes non-random IVs that are not based on constants. A manual review of these applications has shown that many of these programs employ values from external sources, such as user input or third-party libraries. A smaller set used to assemble IVs from device-specific identifiers, such as the IMEI or phone number, and padded unfilled bytes with null values. From a security-critical perspective, these values are not truly random and, thus, should be exchanged with cryptographically strong alternatives.

From our evaluation we can conclude that almost every third manual definition of an IV can be traced back to a constant. The non-automated verification ascertained that our rule is actually capable of identifying statically defined IVs which, in practice, might be wrapped in different data structures, such as byte arrays or strings. Although the resolution of these values is naturally constrained by the capabilities of our backtracking mechanism, the check for the presence of certain object types or APIs usages withstands possible inaccuracy in the inspected data flows. In other words, we prevent possible false positives by strictly concentrating on the occurrence of predefined data types. For example, if the backtracking process would reveal a constant integer value, it would not be considered as a plausible IV. Nevertheless, it is conceivable that an application statically composes a byte array by concatenating individual integer values. The manual application review drew our attention on this peculiarity and enabled us to add detection for this sort of IV.

7.2.3 No Constant Encryption Keys

Retaining encryption keys private is a fundamental requirement to prevent unrelated access to confidential data. Symmetric encryption keys must not be hard-coded in Android applications in order to preserve the meaningfulness of encryption. Applying an automated and manual analysis on 253 applications showed that 18 dispose of statically defined keys.

In total, we found that 141 programs include 454 calls to the `init(...)` methods of the `javax.crypto.SecretKeySpec` API. Considering that distinct instances of `SecretKeySpec` are often used in combination with the `Cipher` API for encryption and decryption, it is reasonable that the concerned applications contain, on average, more than three invocations of this method. Regardless of whether multiple `SecretKeySpec` objects refer to the same key internally, we individually target the provenance of all keys.

Table 7.2 summarizes the findings of our evaluation for this rule. Backtracking every passed `key` parameter shows a rate of 9% or 43 constant keys for symmetric encryption. By also backtracking the second parameter, passed to the `SecretKeySpec` constructor, we were additionally able to determine the algorithm the key is wrapped for. Although public-key cryptography operates with pairs of keys, the byte array of one key can, in principle, be declared as a `SecretKeySpec` object. Since 25 invocations specified RSA as algorithm to use, we also verified whether keys for asymmetric cryptoschemes are stored as constants. From a security-critical perspective, this would be unproblematic if the stored key would correspond to a public key. In contrast, if it be a constantly stored private key instead, it would have to be considered compromised. However, neither the automated nor the manual inspection revealed a statically defined encryption for use with public-key encryption schemes.

SecretKeySpec API key derivation	Count	[%]
Constant symmetric key	43	9%
Constant asymmetric key	0	0%
Mixed: constant + key from variable input	8	2%
Non-static encryption key	403	89%
Total invocations:	454	100%

Table 7.2: Analysis results for encryption keys passed to `SecretKeySpec->init(...)`

We manually found that backtracking the key did not yield an unambiguous result for 10 constructions. Precisely, 8 of the affected `SecretKeySpec` instances are induced mixed keys which consist of a statically defined key that is concatenated with a non-constant input. Interpreting these keys as constants would not reflect the actual situation since we are unable to make assumptions about the entropy provided by the non-constant part. However, it would neither be correct to declare these keys non-static. As a remedy, Table 7.2 outlines these type of findings separately.

The non-automated analysis, furthermore, showed that each invocation refers to a distinct symmetric key. Consequently, the 43 constant symmetric encryption keys also denominate unique keys. While it is trivial to distinguish constant secrets by their value, our framework currently fails to detect situations when multiple data flows are superposing each other. For example, password fields in applications are likely to provide the input for non-constant keys. Now if the same password is supplied to a variety of `SecretKeySpec` instances, our data flow analysis might separately discover the identical key multiple times. Although this issue is not noticeable with individually performed application analysis, it potentially biases statistical evaluation results if identical key references are counted multiple times.

7.2.4 No Constant Passwords or Salts for PBE

Constantly defining a passwords and salt values for Password-Based Encryption (PBE) contradicts the goal of using PBE to derive cryptographically strong keys. Tailored to determine violations of this basic sentence, the evaluation of our security rule revealed that of 253 inspected applications 8 include a constant password and 22 a statically defined salt value.

Searching for invocations of the class `javax.crypto.spec.PBEKeySpec`, we found 88 calls in 47 distinct applications. While it is mandatory to pass a password to the constructor, it depends on the actual method signature whether a salt is provided as secondary argument. Nevertheless, salt values can also be wrapped in objects of the type `javax.crypto.spec.PBEParameterSpec`. Taking both possibilities into account, we found 93 constructions that enclosed salt values for PBE. Surprised by the fact that this number exceeded the amount of defined `PBEKeySpec` constructions with passwords, a manual review clarified that 18 applications use `PBEParameterSpec` in conjunction with the `Cipher` API in order to generate secrets. This leads us to conclude that 75 `PBEKeySpec` instances comprise both passwords and salt values.

During the evaluation, a data flow analysis has been performed for the values that were passed to the inspected APIs. The results are summarized in Table 7.3. As can be seen, backtracking 88 `password` parameters has emitted 14% or 12 statically defined values. For these PBE constructions, the derived encryption key has to be considered broken since the confidentiality of the

JCA provided PBE methods	Passwords		Salt values	
	Count	[%]	**Count**	[%]
PBEKeySpec API	88		75	
PBEParameterSpec API	0		18	
Constant	12	14%	39	42%
Random API	2	2%	7	8%
SecureRandom API	6	7%	17	18%
Other input	68	77%	30	32%
Total invocations:	88	100%	93	100%

Table 7.3: Evaluation results for 47 applications containing PBE method calls

encrypted data is no longer guaranteed. Likewise, 42% or 39 salt values are constantly defined and, thereby, negate the goal of PBE to hinder table-based attacks. Although the number of constant passwords is comparably small, it could be disclosed that 8 invocations specify a hard-coded salt and a statically defined password value.

The cryptographically insecure `Random` API has been employed for the generation of 2 passwords and 7 salt values. Since this PRNG is not designed to provide the required entropy for cryptographic applications, it should be discarded in favour of the `SecureRandom` API. Overall, it can be seen that 74% of the tracked passwords and 50% of the salt values do *not* originate from constants or weak PRNGs. A manual analysis of the corresponding applications revealed that passwords are typically supplied by user input fields. For the origin of non-predictable or non-constant salt values, no predominant source was identifiable.

7.2.5 Not Fewer than 1000 Iterations for PBE

A low iteration count for Password-Based Encryption (PBE) reduces the costs and computational complexity of table-based attacks on derived keys. In accordance with the PKCS#5 standard, our security rule targets PBE constructions that employ less than the minimally advised amount of 1000 iterations. Applied on the previously selected set of 253 applications, our rule identified 35 programs that undercut the defined threshold.

The automated backtracking of `javax.crypto.spec.PBEKeySpec` and `javax.crypto.spec.PBEParameterSpec` API constructors that involve an `iterationCount` argument has uncovered 93 individual invocations in 47 applications. Evidently, this result also conforms with the method calls for the declaration of salt values that have been uncovered with the preceding evaluation. Our security rule succeeded in determining the used iteration count for all invocations. Also confirmed by the manual application review, obviously none of the inspected programs is designed to dynamically adjust the number of iterations.

As depicted in Figure 7.4, a total of 65% or 60 PBE constructions in 35 distinct programs employs less than 1000 iterations. As opposed to that, the remaining 33 instances in 14 distinct applications use at least the minimally recommended count. Regarding the distribution of the values, a significant prevalence of 20 iterations is obvious. While the given figure does not indicate

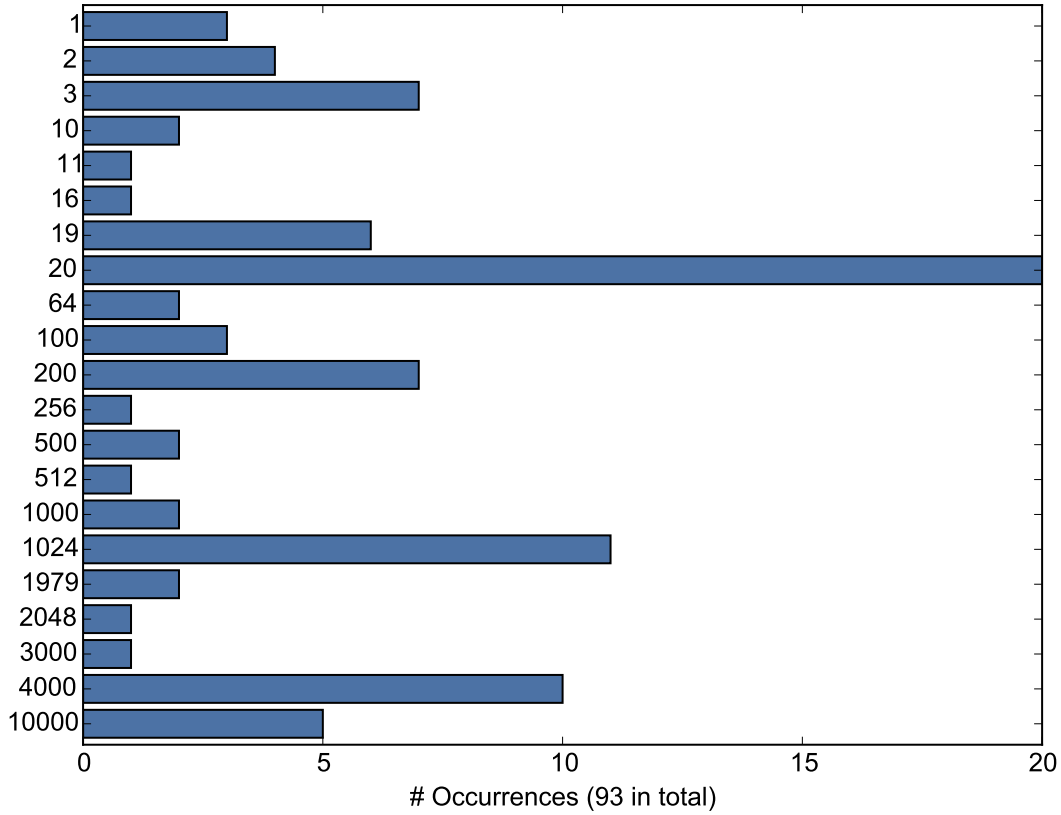

Figure 7.4: Distribution of iterations counts for PBE method calls

the iteration distribution in consideration of distinct applications, a manual analysis shows that this count is shared by a set of 14 programs that belong to different categories and split as follows: 5 Mobile Banking, 6 Password Manager, and 1 Messenger application.

Individually assigned to the category of each investigated program, the listing in Table 7.4 illustrates the distribution of applications with regard to the defined threshold of 1000 iterations. Except for the category Mobile Banking, a significant prevalence for iteration counts undercutting the recommended minimum can be observed.

#	Category	Iteration count distribution	
		$<$ **1000**	$\geq$ **1000**
100	Password Manager	29	18
55	Mobile Banking	8	10
51	Cloud Storage	8	2
38	Messenger	13	2
9	Secure Container	2	1
PBE method calls:		60	33

Table 7.4: Distribution of iterations counts over the investigated dataset

7.2.6 No Static Seeds for SecureRandom

Seeding a pseudorandom number generator (PRNG) with a statically defined value causes the underlying algorithm to produce a deterministic output. Our security rule focusses on invocations of the `java.security.SecureRandom` API and is targeted to uncover constant seed values which are not suited for security-critical applications. The automated analysis of our rule determined 11 problematic seeds in a total of 53 programs that employ the concerned API.

The analysis results highlight that 59 explicitly seeded instances of the `SecureRandom` API can be found in 53 distinct applications. As listed in Table 7.5, a total of 11 constant seeds are passed to the constructor of the API whereas 10 values are derived from statically defined string objects using `String->getBytes()` and a hard-coded byte array in one case. While the automated analysis targeted both the constructor and the `SecureRandom->setSeed(...)` methods, only the first yielded results. Also confirmed by manual review, obviously no application of our dataset intended to set a 64-bits integer as seed.

SecureRandom API seed values	**Count**	[%]
Constant string	10	17%
Constant byte array	1	2%
Constant 64-bits integer	0	0%
Non-static seed	48	81%
Total invocations:	59	100%

Table 7.5: Results for 53 applications defining seed values for `SecureRandom`

The manual inspection of the determined strings suggests that all of them have been chosen deliberately. At least, their value can be reduced to entries in standard dictionaries. What distinguishes these findings from those in previous rules is the fact that the documentation [4] on `SecureRandom` clearly advises to abstain from using a predictable seed since it may result in a severe security issue. A practical remedy is the use of Android 4.2 or higher where a change from the default PRNG provider Apache Harmony to the AndroidOpenSSL mitigated the problem.

Nevertheless, the concept of this rule does not include the possibility that the former behaviour is enforced by explicitly specifying a different provider. Therefore, we would have to backtrack the name of a possibly employed provider and compare it against a list of providers that are known to support the problematic substitution of the entire PRNG state. Since we were unable to observe this scenario in practice, no steps have been taken to adjust the rule for this corner case.

7.2.7 No MessageDigest Without Content

Calculating a hash digest on an empty input message results in a deterministic and predictable output that does not withstand trivial brute-force attacks. Our security rule is designed to analyze data flows and alert if mandatory input values appear to be absent. The automated application on our previously composed dataset found out 84 problematic execution paths in 157 applications that employ the `java.security.MessageDigest` API in order to invoke a particular hash function.

[4] https://developer.android.com/reference/java/security/SecureRandom.html

MessageDigest API usage	**Count**	[%]
Without content	84	22%
With content	302	78%
`digest()` calls	386	67%
Without `digest()` calls	186	33%
Total invocations:	572	100%

Table 7.6: Results for 157 applications using the `MessageDigest` API

As listed in Table 7.6, we disclosed a total of 572 usages of the `MessageDigest` API. Tracking the found objects in forward direction showed that data flows in 67% or 386 instances basically involve a call to the `MessageDigest->digest()` method while another 33% or 186 derive hash digests differently. The subsequent analysis concentrated on the individual execution paths. The data flows of 78% or 302 instances contain invocations of the `MessageDigest->update(...)` methods prior to calling the `digest()` method. For 22% or 84 instances the automated evaluation indicated that at least one execution path declared no message to digest.

False Positives

The manual review confirmed the basic concept of our security rule but also disclosed already known deficiencies in our static slicing implementation (see Section 5.6). Since these constraints led to a multitude of false positives, we will now briefly exemplify two practically occurring situations where the forward tracking mechanism failed to construct the entire data flow.

```
1 iget-object v0, p0, Lorg/apache/http/impl/auth/NTLMEngineImpl$HMACMD5;->md5:
2   Ljava/security/MessageDigest;
3 invoke-virtual {v0, v1}, Ljava/security/MessageDigest;->update([B)V
4 iget-object v0, p0, Lorg/apache/http/impl/auth/NTLMEngineImpl$HMACMD5;->md5:
5   Ljava/security/MessageDigest;
6 invoke-virtual {v0}, Ljava/security/MessageDigest;->digest()[B
```

Listing 7.1: Incomplete `MessageDigest` data flow (reference problem 1)

Assuming that a `MessageDigest` instance is tracked in forward direction, an `iput-object` instruction causes the object reference to be stored into an instance field. Referring to the example provided in Listing 7.1, the subsequent slicing of all individual field accessors will reach the first `iget-object` instruction which loads the reference into the register `v0`. Next, the `update(...)` method is called on the object in `v0`, setting a message that is referenced in `v1`. The problem is now the second `iget-object` instruction which, in fact, *overwrites* the register `v0` with a new reference that actually does not differ from the already contained value. Since the tracking mechanism currently does not inspect the value of tracked registers, we are unable to build a coherent data flow for this example. Another problem is that the second `iget-object` instruction represents a individual field accessor. Hence, no relation to the preceding statement is recognizable and the final `digest()` call would be wrongly identified as being called without content. In order to resolve this issue, profound changes of the slicing mechanism are indispensable, such that a context-sensitive state of tracked registers is held.

```
1 invoke-static {v5}, Ljava/security/MessageDigest;->getInstance
2   (Ljava/lang/String;)Ljava/security/MessageDigest;
3 move-result-object v20
4 move-object/from16 v1, v20
5 invoke-direct {v11, v0, v1}, Ljava/security/DigestOutputStream;-><init>
6   (Ljava/io/OutputStream;Ljava/security/MessageDigest;)V
7 invoke-virtual/range {v20 .. v20}, Ljava/security/MessageDigest;->digest()[B
```

Listing 7.2: Incomplete `MessageDigest` data flow (reference problem 2)

The second example in Listing 7.2 illustrates a problem that is also caused by the absent inspection of references. By invoking the method `getInstance(...)`, a `MessageDigest` instance is derived and the resulting object stored in register `v20`. The subsequent `move-object/from16` instruction copies the object reference from the register `v20` to `v1`. Although our tracking mechanism will continue to follow `v1` and `v20`, it does not take into account that both registers reduce to the same object. In the sample code, tracking the register `v1` reveals that a message input is set using the auxiliary class `DigestOutputStream`. The data flow of the register `v20`, however, terminates with a call to the `digest()` method and does not consider that a content has been assigned via `v1`. A solution for this problem would have to include the detection of object references and a consideration of the order of execution statements.

With regard to the evaluation of this security rule, we can conclude that using the pursued detection strategy it is feasible to determine digest calculations without content. Nevertheless, the assessed data flows should not be tampered with imprecision in order to prevent false positives.

7.2.8 No Password Leaks

The secrecy of sensitive data, such as passwords, must not be impaired by data flows that allow an attacker to learn secret credentials. Therefore, our security rule targets method invocations that substantially affect the security of an entered password. The automated evaluation on our dataset found 1206 input fields for passwords in 253 applications whereas 131 fields appear to process passwords with inappropriate methods.

Our evaluation disclosed XML-based password fields in 167 of 253 distinct applications. As depicted in Figure 7.5, we found that 18% or 220 passwords are processed by security-related APIs. Whether this is advisable for a particular password depends on the individual program and the intention of a password field. For example, Mobile Banking applications which provide only a login functionality for a backend service, are unlikely to need strong cryptography in order to

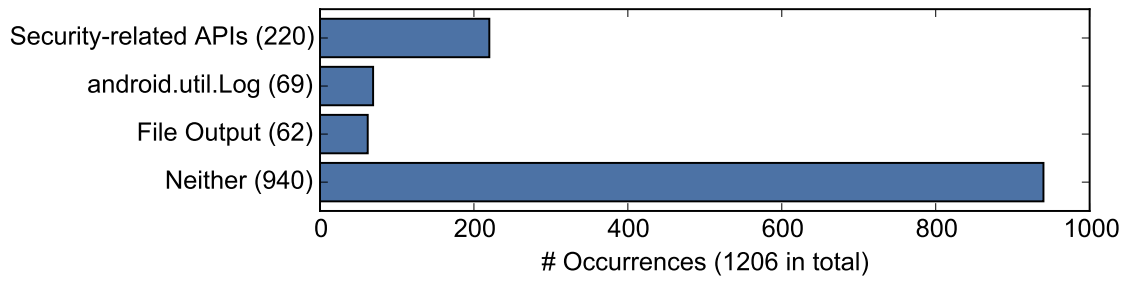

Figure 7.5: Distribution of API usage for password fields

fulfill their purpose. In other words, an application must not a priori be considered insecure if it does not invoke methods of the `javax.security.*` or `javax.crypto.*` packages. Considering the evaluation of a single application whose context might be known, the information whether security-related APIs are called, might be of significant relevance.

Our rule is, furthermore, designed to reveal general misconceptions that substantially affect the secrecy of a password, regardless of the actual application domain. In particular, the automated analysis found out that 6% or 69 of all entered passwords were written to a logging function. Likewise, 5% or 62 passwords appeared to be written to output streams or files. Individually assigned to the category of each program, Table 7.7 predominantly shows that the data flows in 49 of 100 Password Manager applications involve calls to security-related APIs. Finally, the listing also exposes that 23% or 57 programs do not process passwords appropriately.

#	Category	Security-related APIs		android.util.Log		File Output	
		Count	[%]	Count	[%]	Count	[%]
100	Password Manager	49	59%	16	52%	14	54%
55	Mobile Banking	11	13%	3	10%	3	12%
51	Cloud Storage	13	16%	9	29%	4	15%
38	Messenger	9	11%	2	6%	4	15%
9	Secure Container	1	1%	1	3%	1	4%
Total usage by apps:		83	100%	31	100%	26	100%

Table 7.7: Distribution of password usage over the investigated dataset

The precision of our analysis results is strongly linked to the accuracy of the inspected data flow graphs. Aside from known deficiencies in the static slicing concept (see Section 5.6), our manual application review confirmed the expressiveness of the checked methods. Of course, this does not imply that our rule is complete. In fact, additional patterns could be suited to reveal further security misconceptions that impede the security of entered passwords.

7.3 General Discussion of the Results

The goal of our evaluation was to assess the accuracy of the elaborated security rules and to gain an insight into the distribution of security-critical problems. After compiling a dataset of 253 programs, we performed an automated and manual analysis and evaluated the result.

It has shown that the automated analysis using CryptoSlice is capable of uncovering a multitude of substantial weaknesses in applications that process sensitive data. We verified the accuracy of the results by manually reproducing them in affected applications. While this approach enabled us to identify obvious false positives, it did not contribute to the identification of those matches that were not detected by our rules. However, by focussing on misconceptions in specific security-critical constructions, we never intended to disclose ulterior issues.

From the obtained results we conclude that the concept of all security rules qualifies for an application on an arbitrary dataset. Since the output of some rules might not be unambiguous, it seems advisable to remedy the remaining shortcomings of the static slicing mechanism.

Chapter 8

Conclusion

Software vendors are leveraging the great popularity of mobile devices to also deploy applications which process sensitive user data. A wide range of Android applications perform security-critical tasks and require that user inputs are processed reliably. For this to achieve, a correct application of security-critical functionality is indispensable, ensuring that no sensitive data can be leaked within the data flow and that cryptographic systems are applied correctly, in case their use is appropriate.

In this thesis, we presented our new static analysis framework CryptoSlice, which is capable of investigating security-related code in Android applications. Leveraging the ability to trace information in forward and backward direction, the framework can be employed to determine the data flow of relevant code segments. The resulting slice graphs visualize information dependencies and can serve as a basis for further analysis. Built upon an easily extendable architecture, CryptoSlice provides an integrated environment that manages automatically to carry out an entire program inspection. Nevertheless, the framework also facilitates the manual analysis of arbitrary applications by processing generally applicable slicing patterns which generalize traditional slicing criteria. Irrespective of whether data is tracked in forward or backward direction, the resulting slices can be mapped to well-arranged data flow graphs.

Assuming that sensitive information explicitly concerns user-provided data, CryptoSlice pursues a novel approach to follow the trace of a password right from the point where it enters an application. Starting at a definable trigger point, such as a user input field, we compute all statically reachable data flows, which can then be investigated regarding security-related misconceptions.

Our framework includes a series of security rules that are tailored to identify and analyze common implementation flaws. For this thesis, their focus has been put on the inspection of Java Cryptographic Architecture (JCA) provided methods. Since the generic design of the JCA gives developers the ability to configure integral cipher parameters deliberately, a weak parameter selection may drastically impede the originally intended level of security. In order to track down wrongly chosen security attributes, we employ targeted slicing patterns that disclose whether the input is subject to inadequate processing. The presentation of every rule included a conclusive problem statement and explained the chosen strategy for the identification of misuse. Finally, we also reflected on possible shortcomings and discussed alternative approaches.

Applied on a set of selected applications, we evaluated the implemented rules regarding their accuracy and aimed to gather an insight into the distribution of security-critical issues. Using our framework, we performed an automated analysis of all rules and manually verified the findings. Resulting in an empirical study, we disclosed a plenitude of critical problems that substantially undermine the security of the affected applications. Aside from confirming the meaningfulness

of the elaborated stategies of all rules, the automated and manual evaluation also pointed out manageable deficiencies in our static slicing implementation.

The modular architecture of CryptoSlice supports the extension with further functionality and analysis capabilities. Among other possibilities, the following list briefly denotes ideas for future work that would still enhance the framework:

- **Additional security rules to detect misuse**
  Aside from analyzing the data flow of password fields, for this thesis we have concentrated on the potential misconceptions of APIs that are provided by the JCA. Although the presented rules cover a broad set of occasions, they consider only a subset of possible misuse. For example, the security of RSA would be compromised if the private exponent is hard-coded. After elaborating a concrete strategy, this problem could be verified in another rule.

  Alternatively, it would also be feasible to target other security issues, such as constantly defined credentials for HTTP Basic Authentication, broken TLS certificate validation, or requests of applications to gain superior privileges.

- **Rules to generally assess applications**
  Rather than focussing on implementation weaknesses, rules could also be employed to simply extract interesting information of inspected programs. For example, statically defined passwords for keystores or encrypted databases could be revealed. Similarly, they could be used to fingerprint provided versions of third-party libraries or to list invocations of native methods and a reference to their location in the appendant native library.

- **Establish context-sensitive tracking with references**
  The current tracking mechanism handles registers as individual values. However, this approach does not recognize references to a single object in multiple registers. Resolving this issue would augment the accuracy of the derived data flow graphs. At the same time, it would enable the identification of individual object instances.

- **Implement a look-ahead mechanism for static slicing**
  At the moment, CryptoSlice holds the registers to track in a FIFO queue. If registers would be inserted by their correspondance with the slicing criterion, a more targeted slicing process would be possible. Of course, unless configured differently, this action might have an impact on the execution time but would not change the slicing results.

- **Process efficiency**
  Currently, the analysis workflow immediately transforms Smali code into an object-based representation. Depending on the size of an application this may require a considerable amount of memory. As a remedy, code could be loaded on demand.

With CryptoSlice an easily adaptable framework for the static analysis of Android applications is provided. By applying static slicing in forward and backward direction, we are able to uncover the misuse of security-critical APIs and to arbitrarily assess the data flow of passwords.

Bibliography

[1] Martin Abadi and Bogdan Warinschi. “Password-based encryption analyzed”. In: *Automata, Languages and Programming* (2005), pages 1–12.

[2] Carlisle Adams et al. “On The Security of Key Derivation Functions”. In: *Information Security*. Edited by Kan Zhang and Yuliang Zheng. Volume 3225. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pages 134–145. ISBN 978-3-540-23208-7. doi:10.1007/978-3-540-30144-8_12.

[3] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

[4] Moutaz Alazab et al. “Analysis of Malicious and Benign Android Applications”. In: *International Conference on Distributed Computing Systems Workshops* (2012), pages 608–616. doi:10.1109/ICDCSW.2012.13.

[5] Leonid Batyuk et al. “Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications”. In: *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, 2011, pages 66–72. ISBN 978-1-4673-0034-6. doi:10.1109/MALWARE.2011.6112328.

[6] Andrey Belenko and Dmitry Sklyarov. “„Secure Password Managers “and „Military-Grade Encryption “on Smartphones: Oh, Really”. In: *Blackhat Europe* (2012). `http://mx5.elcomsoft.com/WP/BH-EU-2012-WP.pdf`.

[7] Jean-Francois Bergeretti and Bernard A. Carré. *Information-flow and data-flow analysis of while-programs*. 1985. doi:10.1145/2363.2366.

[8] Lily Chen. *Recommendation for key derivation using pseudorandom functions*. Technical report October. 2009. `http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf`.

[9] Manuel Egele et al. “An empirical study of cryptographic misuse in android applications”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS ’13*. ACM Press, 2013, pages 73–84. ISBN 9781450324779. doi:10.1145/2508859.2516693.

[10] William Enck and Damien Octeau. “A Study of Android Application Security.” In: *USENIX security ...* August (2011), pages 21–21. ISSN 0364-2348. doi:10.1007/s00256-010-0882-8. `http://www.usenix.org/event/sec11/tech/slides/enck.pdf`.

[11] William Enck et al. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *Osdi ’10* 49 (2010), pages 1–6. ISSN 03601315.

[12] Sascha Fahl, Marian Harbach, and Marten Oltrogge. “Hey, You, Get Off of My Clipboard”. In: ... *Cryptography and Data ...* (2013). http://link.springer.com/chapter/10.1007/978-3-642-39884-1%5C_12.

[13] Sascha Fahl et al. “Rethinking SSL development in an appified world”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. ACM Press, 2013, pages 49–60. ISBN 9781450324779. doi:10.1145/2508859.2516655.

[14] Sascha Fahl et al. “Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security Categories and Subject Descriptors”. In: *ACM Conference on Computer and Communication Security (CCS)*. 2012, pages 50–61. ISBN 9781450316514. doi:10.1145/2382196.2382205.

[15] Christian Fritz and Steven Arzt. “Highly precise taint analysis for android applications”. In: *EC SPRIDE, TU ...* (2013). http://www.informatik.tu-darmstadt.de/fileadmin/user%5C_upload/Group%5C_CASED/Publikationen/TUD-CS-2013-0113.pdf.

[16] Adam P Fuchs, Avik Chaudhuri, and Jeffrey Foster. “SCanDroid : Automated Security Certification of Android Applications”. In: *Read* 10 (2010), page 328. doi:10.1.1.164.6899.

[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*. ACM Press, 2012, page 38. ISBN 9781450316514. doi:10.1145/2382196.2382204.

[18] Clint Gibler et al. *AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale*. 2012. doi:10.1007/978-3-642-30921-2_17.

[19] Ben Gruver. *Smali*. 2014. https://code.google.com/p/smali.

[20] Mary Jean Harrold and Mary Lou Soffa. *Efficient computation of interprocedural definition-use chains*. 1994. doi:10.1145/174662.174663.

[21] Johannes Hoffmann et al. “Slicing droids: program slicing for smali code”. In: *Symposium on Applied Computing*. 2013, pages 1844–1851. ISBN 9781450316569.

[22] Susan Horwitz, Thomas Reps, and David Binkley. *Interprocedural slicing using dependence graphs*. 2004. doi:10.1145/989393.989419.

[23] Burt Kaliski. “RFC 2898 PKCS #5: Password-Based Cryptography Specification Version 2.0”. In: *IETF RFC* (2000), pages 1–35.

[24] Jinyung Kim et al. “Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications”. In: *IEEE Workshop on Mobile Security Technologies (MoST)*. 2012.

[25] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. “Predictability of Android OpenSSL's pseudo random number generator”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. New York, New York, USA: ACM Press, 2013, pages 659–668. ISBN 9781450324779. doi:10.1145/2508859.2516706. http://dl.acm.org/citation.cfm?doid=2508859.2516706.

[26] Bogdan Korel and Janusz Laski. “Dynamic program slicing”. In: *Information Processing Letters* 29.3 (Oct. 1988), pages 155–163. ISSN 00200190. doi:10.1016/0020-0190(88)90054-3.

[27] Christopher Mann and Artem Starostin. “A framework for static detection of privacy leaks in android applications”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC ’12* (2012), page 1457. doi:10.1145/2245276.2232009.

[28] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Volume 106. 1997, page 780. ISBN 0849385237. doi:10.1.1.99.2838.

[29] Andreas Moser, Christopher Kruegel, and Engin Kirda. “Limits of Static Analysis for Malware Detection”. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* (2007). ISSN 1063-9527. doi:10.1109/ACSAC.2007.21.

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

[31] Damien Octeau, William Enck, and Patrick McDaniel. *The ded Decompiler*. Technical report. Network and Security Research Center, 2010. `http://siis.cse.psu.edu/ded/papers/NAS-TR-0140-2010.pdf`.

[32] Damien Octeau, Somesh Jha, and Patrick McDaniel. “Retargeting Android Applications to Java Bytecode”. In: *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*. 2012. `http://siis.cse.psu.edu/dare/papers/octeau-fse12.pdf`.

[33] Oracle. *jarsigner - JAR Signing and Verification Tool*. 2014. `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html`.

[34] Karl J. Ottenstein and Linda M. Ottenstein. *The program dependence graph in a software development environment*. 1984. doi:10.1145/390011.808263.

[35] Colin Percival. “Stronger key derivation via sequential memory-hard functions”. In: *Self-published* (2009), pages 1–16.

[36] Sebastian Poeplau et al. “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications”. In: *Symposium on Network and Distributed System Security (NDSS)*. February. 2014, pages 23–26. ISBN 1891562355.

[37] Niels Provos and David Mazieres. “A Future-Adaptable Password Scheme”. In: *USENIX Annual Technical Conference, ...* (1999), pages 1–12.

[38] Dang Quynh. “Recommendation for Applications Using Approved Hash Algorithms”. In: *NIST Special Publication* (2012).

[39] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks”. In: *Symposium on Network and Distributed System Security (NDSS)*. February. 2014, pages 23–26. ISBN 1891562355.

[40] Vaibhav Rastogi, Yan Chen, and William Enck. “AppsPlayground : Automatic Security Analysis of Smartphone Applications”. In: *CODASPY ’13 Proceedings of the third ACM conference on Data and application security and privacy*. 2013, pages 209–220. ISBN 9781450318907.

[41] Golam Sarwar et al. “On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices”. In: *SECRYPT 2013, 10th International Conference on Security and Cryptography*. Edited by P. Samarati. ACM, 2013, pages 1–15. `http://www.nicta.com.au/pub?id=6865`.

[42] B. Schwarz, S. Debray, and G. Andrews. “Disassembly of executable code revisited”. In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* (2002). ISSN 1095-1350. doi:10.1109/WCRE.2002.1173063.

[43] Marc Stevens. “New collision attacks on SHA-1 based on optimal joint local-collision analysis”. In: *Advances in Cryptology–EUROCRYPT 2013* (2013).

[44] Peter Teufl et al. “Android Encryption Systems”. In: *International Conference on Privacy & Security in Mobile Systems*. 2014.

[45] The Android Open Source Project. *Intents and Intent Filters*. 2014.

[46] Frank Tip. “A Survey of Program Slicing Techniques”. In: *Journal of Programming Languages* 5399.3 (1995), pages 1–65. doi:10.1.1.43.3782.

[47] Meltem Sönmez Turan et al. *Recommendation for Password-Based Key Derivation: Part 1: Storage Applications*. Technical report. Gaithersburg, MD, United States, 2010. `http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf`.

[48] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (1984). ISSN 0098-5589. doi:10.1109/TSE.1984.5010248.

[49] Tao Xie, Fanbao Liu, and Dengguo Feng. “Fast Collision Attack on MD5”. In: (2013), pages 1–12. doi:10.1.1.301.4421.

[50] Baowen Xu et al. “A brief survey of program slicing”. In: *SIGSOFT Softw. Eng. Notes* 30.2 (2005), pages 1–36. ISSN 0163-5948. doi:10.1145/1050849.1050865.

[51] Jianjun Zhao. “Slicing concurrent Java programs”. In: *Proceedings Seventh International Workshop on Program Comprehension* (1999). ISSN 1092-8138. doi:10.1109/WPC.1999.777751.