Πανεπιστήμιο Πειραιώς – Τμήμα Ψηφιακών Συστημάτων
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Τεχνοοικονομική Διοίκηση και Ασφάλεια Ψηφιακών Συστημάτων»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	An Empirical Evaluation of Cryptography usage in Android Applications
Ονοματεπώνυμο Φοιτητή	**Χατζηκωνσταντίνου Αλεξία**
Πατρώνυμο	**Κωνσταντίνος**
Αριθμός Μητρώου	MTE1335
Επιβλέπων	**Δρ. Χριστόφορος Νταντογιάν**

Ημερομηνία Παράδοσης **Ιανουάριος 2014**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Όνομα Επώνυμο Βαθμίδα	Όνομα Επώνυμο Βαθμίδα	Όνομα Επώνυμο Βαθμίδα

Table of Contents

Abstract

It is fact that Cryptography is already highly used in applications involving sensitive data, for example, credentials, credit card pins and other personal information. Nonetheless, developers frequently lack specialized cryptography knowledge. In this master thesis, we analyze 49 android applications in order to determine -via static and dynamic analyses- the specific cryptographic misuses that result in data breaches. The results show that the 85, 71% of the applications examined (i.e. 42 out of the 49 applications) make at least one cryptographic misuse. Thus, we suggest a list of cryptographic rules that are based on our analyses in order to improve the overall cryptographic security in Android applications.

Introduction

It is common knowledge that the need of humans to share publicly information in a manner that would be understandable to only specific group of persons has been existing since million years before computer's invention and establishment. The existence of algorithms akin to Caesar's Cipher proves that the contemporary cryptography has its origins in Caesar's era, as attempts to achieve information security had already begun to take place. Thus, a great progress has been made in the field of cryptography ab aeterno.

As a matter of fact, the imperative need for employing encryption in applications involving sensitive information is already granted in the information technology community, although the utilization of proper and correct practices has not yet been consolidated. The plethora of resources in modern societies has resulted in turning application developing to a widespread tendency. Nevertheless, the developers rarely specialize in the very specific knowledge of information security, even in the case of professionals, as information science constitutes a quite general range. As a consequence, incidents of data breaching and disclosure are very frequent.

Despite the fact that information security is a continual conflict between the benevolent and malevolent, where the former on an attempt to protect sensitive information build more and more secure cryptographic algorithms, and the latter always invent a manner to bypass the security introduced by these algorithms, it is of utmost importance to make good cryptographic practices widely known and adopted in the circles of developers.

Regarding the academic activity in the specific domain, a lot of research has been conducted and many studies have been realized, others constituting a milestone in information security domain, and others introducing significant knowledge in the same field. However, none of them have yet concentrated a set of accepted and not accepted practices so as to form a methodology for the common developers who do not specialize in information security and cryptography, as each work aims at giving prominence to the specific cryptographic mistakes of the applications and not at teaching the developers.

Hence, this master thesis' intention is to eliminate this lack of a complete list of cryptographic misuses and best practices regarding cryptography. In the interest of accomplishing this aim, we base our work on both literature survey and on research conducted especially on android applications, attempting to take into consideration also the prevalent tendencies of developers.

1. Android framework

1.1 Android in General

Android is the mobile operating system produced by Google. Android is a Linux-based software system, and similar to Linux, is a free and open source software, meaning that other companies can use the Android operating system, even in their mobile devices. The original creators were Android Inc. — directed by Andy Rubin, who became the head of Android development at Google after the acquisition in 2005. Google bought the company because they thought Android Inc. had an interesting and important concept of creating a powerful, yet free, mobile operating system. Android helped Google to reach a younger audience as well as give the company a number of brilliant employees from Android Inc. [2]

With a user interface based on direct manipulation, Android is designed to be installed generally on touch screen mobile devices such as Smartphones and tablet computers, with specialized user interfaces for televisions (Android TV), cars (Android Auto), and wrist watches (Android Wear). The OS uses touch inputs that loosely correspond to real-world actions, for example swiping, tapping, pinching, and reverse pinching to manipulate on-screen objects, and a virtual keyboard. Despite being primarily designed for touch screen input, it also has been used in game consoles, digital cameras, regular PCs (e.g. the HP Slate 21) and other electronics. [4]

As already mentioned, Android most commonly comes installed on a variety of Smartphones and tablets from a host of manufacturers offering users access to Google's own services, for example Search, YouTube, Maps, Gmail and more. Namely, the holder of an Android Smartphone is capable of easily searching for information or directions on the web, watch videos, write email and realize many more activities that up until now were executed exclusively on computers.

Android phones are highly customizable devices. Thus, they can be altered to suit a great variety of tastes and needs with background images (known also as wallpapers), themes and launchers which completely change the look of Smartphone's interface. Besides, Android supports for their users the great expediency of downloading applications (free or priced) from the Google Play store (formerly the Android Market) to do an immense variety of activities suchlike logging in Social Media (e.g. Facebook and Twitter), manage bank accounts, play games (e.g. the famous games Angry Birds and Cut the Rope), and save notes.

There are also cloud-based applications which allow users to plan events on from the phone's calendar and see them on their computer or browse websites on their desktop and pick them up on the phone. Camera applications also constitute a popular category, allowing users to take pictures with artistic effects and filters, as well as music players with the use of which the users stream music from the web and create playlists. As for the customization applications, Android users are capable of changing the appearance of their Android handset with numerous wallpapers based on pictures taken by them or downloaded from the internet too. There are also various on-screen widgets to download, which allow access to the settings on the phone, permitting also their alteration. Android users can now create their own system of shortcuts and menus to better suit how they uniquely use the phone.

The majority of applications can be downloaded from the Google Play store (the equivalent of Apple's App Store), which includes a combination of free as well as 'premium' application that users have to pay for. Some applications have a 'lite' version which are free and a 'premium' version, which is an upgrade of the lite version. Others - like Angry Birds - are free, but include adverts or the ability to make in-app purchases. Last but not least, although there are over 1.3 million applications available to Android users in the Google Play store, some developers choose to make their applications available to download from their own sites or alternative application stores. It is fact that users by downloading applications outside of the Google Play store, they run the risk of attack in the form of data theft and make their devices susceptible to viruses.

Another significant feature of Android operating system is that it offers to their users the advantage of connecting the device into a Google Account. The significance of the specific feature is of utmost importance as it is possible for the device to automatically back up user's contacts and sync them to the Google Account, so as to –in the case that the phone is lost or stolen- gain access in the lost data.

However, apart from saving contacts, Syncing is also used in order to keep all types of information, for example websites, calendar entries, files and applications' content updates. This can happen over the phone's mobile data or Wi-Fi connection, seamlessly, in the background. [1]

Android is the most widely used mobile OS and, regarding 2013, the highest selling OS overall. Android devices –according to statistics – have been sold more than Windows, iOS, and MAC OS X devices combined, with sales in 2012, 2013 and 2014 being close to the installed base of all PCs. As of July 2013 the Google Play store has had over 1 million Android apps published, and over 50 billion apps downloaded. A developer survey conducted in April–May 2013 found that 71% of mobile developers develop for Android. At Google I/O 2014, the company revealed that there were over 1 billion active monthly Android users, up from 538 million in June 2013.

Android is popular with technology companies which require a low-cost and customizable operating system for high-tech devices, not needing to be specifically configured by the user before it can be used. The fact that Android is open source has encouraged a large community of developers and enthusiasts to use the open-source code as a foundation for community-driven projects, which add new features for advanced users or bring Android to devices which were officially released running other operating systems. The operating system's success has made it a target for patent litigation as part of the so-called "Smartphone" between technology companies. [4]

Regarding Android's biggest competitors, they are both Apple and Windows phones. Windows Phone has slowly grown into a reputable mobile ecosystem, producing reliable devices, even if they haven't been as popular as Apple and Android phones. On the other hand, Apple jumpstarted both the Smartphone and tablet industries when they released the iPhone in 2007 and the iPad in 2010. Both devices have spurred subsequent products that are not only ameliorated, but get more popular as well. Android may have a better market share worldwide and in the U.S., but Android also has a large number of devices. In most years, Apple releases only one to two iPhones and 3 iPad models, at maximum. Taking those factors into consideration, the case could be made that Apple is doing better in the mobile world than Android, despite what the statistics show. Most analysts agree that Apple is the leader in the Smartphone and tablet market, while Android trails as a close second.

Nevertheless, there is one main difference between Apple's mobile operating system — iOS — and Google's Android: while Android's is open sourced and free for other companies, Apple's iOS is extremely closed source. The specific fact means that it's very limiting in terms of customizing applications. For instance, on iOS, users are not able to change their default web browser from Safari to Google Chrome. Although the specific issue may seem minuscule, it indicates a general lack of iPhone in allowing users to customize both settings and appearance. The open vs. closed operating system debate has been argued time and time again, but in the end, it depends on users' personal preference regarding choosing between a device easily customizable but more complicated (i.e. Android) and a device that is easy to use, but not giving virtually any freedom (Apple's iOS). [2]

Figure 1: Android vs. Windows & iPhone [3]

1.2 Android Versions

Being adherent to the open-source's characteristics, Android is also known for its various versions. The version history of the Android mobile operating system began with the release of the Android beta in

November 2007. However, the first commercial version, Android 1.0, was released in September 2008. Android is under ongoing development by Google and the Open Handset Alliance (OHA), and has seen a number of updates to its base operating system since its initial release.

Since April 2009, Android versions have been developed under a confectionery-themed code name and released in alphabetical order. The exceptions are Android versions 1.0 and 1.1 as they were not released under specific code names:

- Alpha (1.0)
- Beta (1.1)
- Cupcake (1.5)
- Donut (1.6)
- Eclair (2.0–2.1)
- Froyo (2.2–2.2.3)
- Gingerbread (2.3–2.3.7)
- Honeycomb (3.0–3.2.6)
- Ice Cream Sandwich (4.0–4.0.4)
- Jelly Bean (4.1–4.3.1)
- KitKat (4.4–4.4.4)
- Lollipop (5.0–5.0.1)

The most recent major Android update was Lollipop 5.0, which was released on November 3, 2014 along with the Nexus 6 Smartphone, Nexus 9 tablet, and Nexus Player set-top box. [5]

In the following paragraph is cited a quick primer on the different versions of Android that are still in use, for the oldest to the newest.

Android 1.5 – Cupcake

Cupcake was the first major overhaul of the Android OS. The Android 1.5 SDK - released in April 2009 – has brought along plenty of UI changes, the biggest probably being support for widgets and folders on the home screens.

There were plenty of changes behind the scenes, too. Cupcake brought features like improved Bluetooth support, camcorder functions, and new upload services like YouTube and Picasa.

Android 1.5 ushered in the era of the modern Android phone, and the explosion of devices included favorites like the HTC Hero/Eris, the Samsung Moment, and the Motorola Cliq. [6]

Android 1.6 – Donut

Donut, released in September 2009, supported the features that came with Android 1.5 and expanded them. While not being very rich in changes regarding user interface, Android 1.6 made some major improvements behind the scenes, and provided the framework base for the amazing features to come. To the end user, the two biggest changes would have to be the improvements to the Android Market, and universal search.

Behind the screen, Donut brought support for higher resolution touch screens, much improved camera and gallery support, but certainly most importantly, native support for Sprint and Verizon phones. Without the technology in Android 1.6, there would be no Motorola Droid X or HTC Evo 4G.

The devices released with Android 1.6 cover a wide range of taste and features, including the Motorola Devour, the Garminphone, and the Sony Ericsson Xperia X10. [6]

Android 2.0-2.01-2.1 - Eclair

Eclair was a pretty major step up over its predecessors. Introduced in late 2009, Android 2.0 first appeared on the Motorola Droid, bringing improvements in the browser, Google Maps, and a new user

interface. Google Maps Navigation also was first imported in Android 2.0, quickly bringing the platform on par with other stand-along GPS navigation systems.

Android 2.0 quickly upgraded to 2.0.1, which the Droid received in December 2009, mainly bringing bug fixes. And to date, the Droid remains the only phone to have explicitly received Android 2.0.1.

The now-defunct Google Nexus One was the first device to receive Android 2.1 when it launched in January 2010, bringing a UI with cool 3D-style graphics. From there, the end of Android 2.1 has been relatively slow. Manufacturers skipped Android 2.0 in favor of the latest version but needed time to tweak their customizations, such as Motorola's Motoblur.

HTC's Desire and Legend phones launched with Android 2.1 later in the year, bringing a new and improved Sense user interface. [6]

Android 2.2 - Froyo

Android 2.2 was announced in May 2010 at the Google IO conference in San Francisco. The single yet significant change was the introduction of the Just-In-Time Compiler (JIT) which significantly speeds up the phone's processing power.

Along with the JIT, Android 2.2 also brings support for Adobe Flash 10.1. That means that the users could play Flash-based games in Android's web browser, in opposition to the iPhone.

Froyo also brought native support for tethering, meaning that Android users could use their Smartphone’s data connection to provide Internet (wirelessly or with a USB cable) to just about any device selected. Sadly, most carriers will strip this native support in exchange for some sort of feature they can charge for. [6]

Android 2.3 - Gingerbread

Android 2.3 came out of the oven in December 2010, and like Eclair, has a new "Google phone" to go along with -- the Nexus S. Gingerbread brought a few UI enhancements to Android, things like a more consistent feel across menus and dialogs, and a new black notification bar, but still looks similar and feels like the Android that Android users were used to, with the addition of a slew of new language support.

Gingerbread brings support for new technology as well. NFC (Near Field Communication) is now supported, and SIP (Internet calling) support is now native on Android. Further optimizations have been made for better battery life.

Behind the scenes, JIT (the Just-In-Time compiler) optimizations had taken place, and great improvements had been made to Androids garbage collection, which should eliminate any stuttering and improve UI smoothness. Furthermore, a new multi-media framework had been introduced in order to succeed better support of sound and video files. [6]

Android 3.X - Honeycomb

Android 3.0 came out in February 2011 with the Motorola Xoom. The specific version constitutes the first version of Android specifically made for tablets, and brings a lot of new UI elements to the table. Elements such a new System bar at the bottom of the screen to replace the Status bar have been added on phones, as well as a new recent applications button.

Some of the standard Google applications have also been updated for use with Honeycomb, including the Gmail app and the Talk app. Both make great use of fragments, and the Talk app has video chat and calling support built in. Under the hood, 3D rendering and hardware acceleration have been greatly improved. [6]

It is also important to mention that Honeycomb Android version was the first to offer its users the option to deploy full disk encryption (FDE). Android’s FDE offering then remained largely unchanged until Google fortified it in Android 4.4.

This initial encryption scheme deployed by Google in Android was apparently quite secure. However, its implementation in the software, as is so often the case, is where weaknesses arise. Particularly, the security of this encryption scheme depends almost entirely on the complexity of the disk encryption passphrase and its susceptibility to brute-force attacks. In other words, the strength of the disk encryption on Android was as strong (or weak) as the lock-screen password selected. And in most cases it is really weak as people tend to set short lock-screen passwords.

A built-in rate limiting mechanism made brute-force attacks impractical because an attacker would be locked out after a certain number of login attempts. There is a way of brute-forcing weak PINs or passwords in order to decrypt content stored on Android devices.

Again, this attack would be much more difficult to perform against a strong password. But if a typical 4 to 6 character password is utilized, then users' data can be decrypted literally in only a few seconds. [19]

Honeycomb also shows Google's new distribution method, where manufacturers are given the source code and license to use it only after their hardware choices have been reviewed and approved by Google. This hinders third party development, as the source code is no longer available for all to download and build, but Google assures the public that they will address this issue in the future.

Improvements to Honeycomb were announced at Google IO in May 2011 as Android 3.1, and Android 3.2 has followed. [6]

Android 4.0 – Ice Cream Sandwich

The follow-up to Honeycomb was announced at Google IO in May 2011 and released in December 2011. Dubbed Ice Cream Sandwich and finally designated Android 4.0, Ice Cream Sandwich brings many of the design elements of Honeycomb to Smartphones, while refining the Honeycomb experience.

The first device to launch with ICS was the Samsung Galaxy Nexus. The Motorola Xoom and the ASUS Transformer Prime were the first tablets to receive updates, while the Samsung Nexus S was the first Smartphone to make the jump to Android 4.0. [6]

Android 4.1-4.3 – Jelly Bean

Jelly Bean arrived at Google IO 2012, with the release of the ASUS Nexus 7, followed by a short update for unlocked Galaxy Nexus phones. Later in the year, the release of the Nexus 10 and Nexus 4 updated Android from 4.1 to 4.2 and on to 4.3, but the version remained Jelly Bean. The release polished the UI design started in Ice Cream Sandwich, and brought a plethora of new features to the table.

Besides the new focus on responsiveness with Project Butter, Jelly Bean introduces multi-user accounts, actionable notifications, lock screen widgets, quick-settings in the notification bar, Photosphere to the "stock" Android camera and Google Now.

Jelly Bean is hailed by many as the turning point for Android, where all the great services and customization options finally meet great design guidelines. It's certainly quite visually pleasing, and we'd argue that it's become one of the nicest looking mobile operating systems available. [6]

Android 4.4 – KitKat

In the specific Android Version, Google moved towards a stronger crypto-system than that employed in Honeycomb version. Despite this, it is still based on a PIN or password, thus, it is still possible to perform an attack and ultimately brute-force weak PINs and passwords, though in Android 4.4 it took a matter of minutes rather than seconds.

Some varieties of Android 4.4 allowed users to create a separate encryption password. In this way the barrier to entry was increased, because users with separate encryption passwords were protected by two barriers (lock-screen and crypto passwords). [19]

Android 5.0 – Lollipop

Google has announced Android 5.0 Lollipop, and it ushers is a new design language and support for 64-bit devices. Google also for the first time has provided developer beta previews of the software, so that the popular to Android users applications can be ready when the new version drops.

In Lollipop version big changes take place under the hood, and a plethora of new API changes in addition to forward-facing features like a new interface. Google plans to update their own Nexus 5, Nexus 4 and Nexus 7 to Android L, and other companies like Motorola have affirmed their intentions to do the same. [6]

The most important difference to the previous versions however, is the fact of turning FDE (Full Disk Encryption) on by default for the first time. In addition to this, Android L is expected to include hardware protection for keys used in disk encryption, as well as hardware acceleration for encrypted disk access. What is more, the attacks that threatened the older versions of android crypto system do not work for Android L. The exact reason for that is unclear, due to the fact that there is not yet any available source-code for the operating system. Last but not least, encryption key derivation is no longer based purely on a user's passphrase, PIN or lock-screen password. Instead, it seems decryption in future varieties of Android will be based only in part on the user's lock-screen PIN or password. [19]

1.3 Android Memory Management

Memory is one of the main resources of any computing device. As technology has progressed, memory capacity has increased for various devices, making retrieval of data one of the parameters of speed and efficiency within the computing device.

There are two kinds of memory, the permanent storage and the Random Access Memory (RAM). RAM is a repository for all data and instructions for programs that are running and is used by the system to load, execute and manipulate key parts of the operating system, applications or data and is not saved on reboot.

It has the values required for the computing device to actually function, such as passwords, encryption keys, usernames, app data and data from system processes and services. Therefore, data movement to and from the RAM has o direct impact on the processing speed of the device. RAM is a valuable resource in any software development environment, but it is even more valuable on a mobile operating system where physical memory is constrained.

Moreover, memory on mobile devices can be distinguished between internal and external memory. Nowadays, mostly flash memory, consisting of NAND or NOR types, is used for internal memory, while external memory devices are SIM cards, SD cards, MMC cards, CF cards and memory sticks.

NAND flash memory is a type of non-volatile storage technology and does not require power to retain data. One of the important goals of NAND flash development has been to reduce the cost per bit and increase maximum chip capacity so that flash memory can compete with magnetic storage devices, suchlike hard disks.

NAND has a finite number of write cycles. NAND failure is usually gradual as individual cells fail and overall performance degrades.

Android is a Linux-based operating system, that includes the middleware and key applications for its tasks to be handled, and so on, memory management is handled by the native Linux kernel. Android offers several options for application data storage. The chosen solution depends on whether the data should be private to a specific application or accessible to other applications and how much space the data requires.

Android provides many kinds of storage for applications to store their data. The storage places are shared preferences, internal and external storage, SQLite storage, and storage via network connection. A content provider is available, in order to even give access to all private data from other applications, which is an optional content that exposes read/write access to the application data.

Android does not offer swap space for memory. Nonetheless, it utilizes paging and memory-mapping (mmapping) to manage memory. This means that any memory that the user modifies –whether by allocating new objects or touching mmapped pages- remains resident in RAM and cannot be paged out.

Generally, Android devices store an enormous amount of data, typically combining both personal and work data. Applications are the primary source of this data and there are a number of sources for applications, including:

- Applications that ship with Android
- Applications installed by the manufacturer
- Applications installed by the wireless carrier
- Additional Google/Android applications
- Applications installed by the user, typically from the Android Market

1.4 Android Applications Structure

The structure of an Android application consists of four different components, which are: *Activity*, *Service*, *BroadcastReceiver* and *ContentProvider*. An application does not necessarily contain all four of these components, but to present a graphical user interface there has to be at least an Activity.

Applications can start other applications or specific components of other applications by sending an Intent. These Intents contain, apart from other things, the name of desired action executed. The IntentManager resolves incoming intents and starts the proper application or component. The reception of an Intent can be filtered by an application.

Services and broadcast receivers allow applications to perform jobs in the background and provide additional functionality to other components. Broadcast receivers can be triggered by events and only run a short period of time whereas a service may run a long time. The compiled code of the application components and additional resources like libraries, images and other necessary data is concentrated and packed into a single .apk file that forms the executable Android application. [7]

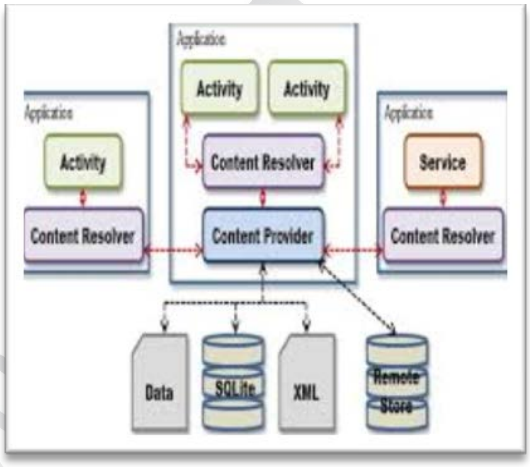

Figure 2: Android Applications Structure [9]

1.4.1 Activity Lifecycle

An Activity is an application component that provides a screen with which the users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is "connected" with a specific window in which it draws its user interface. An application usually consists of multiple activities that are loosely interrelated to each other. Typically, one activity in an application is specified as the main activity, which is presented to the user when launching the application for the first time. Each activity can then initiate another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the "back stack" and takes user focus. The back stack abides to the basic "last in, first out" stack mechanism, so, when the user is done with the current activity and presses the *Back* button, it is popped and destroyed from the stack and the previous activity resumes.

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

Activities in the system are managed as an activity stack. When a new activity is started, it is placed on the top of the stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits.

An activity has essentially four states:

- If an activity in the foreground of the screen (at the top of the stack), it is active or running.
- If an activity has lost focus but is still visible (that is, a new non-full-sized or transparent activity has focus on top of your activity), it is paused. A paused activity is completely alive (it maintains all state and member information and remains attached to the window manager), but can be killed by the system in extreme low memory situations.
- If an activity is completely obscured by another activity, it is stopped. It still retains all state and member information, however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
- If an activity is paused or stopped, the system can drop the activity from memory by either asking it to finish, or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

The diagram depicted in Figure 6 shows the important state paths of an Activity. The square rectangles represent callback methods you can implement to perform operations when the Activity moves between states. The colored ovals are major states the Activity can be in.

There are three key loops you may be interested in monitoring within an activity:

- The entire lifetime of an activity happens between the first call to onCreate(Bundle) through to a single final call to onDestroy(). An activity will do all setup of "global" state in onCreate(), and release all remaining resources in onDestroy(). For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate() and then stop the thread in onDestroy().
- The visible lifetime of an activity happens between a call to onStart() until a corresponding call to onStop(). During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user. For example, you can register a BroadcastReceiver in onStart() to monitor for changes that impact your UI, and unregister it in onStop() when the user no longer sees what you are displaying. The onStart() and onStop() methods can be called multiple times, as the activity becomes visible and hidden to the user.
- The foreground lifetime of an activity happens between a call to onResume() until a corresponding call to onPause(). During this time the activity is in front of all other activities and interacting with the user. An activity can frequently go between the resumed and paused states -- for example when the device goes to sleep, when an activity result is delivered, when a new intent is delivered -- so the code in these methods should be fairly lightweight.

The entire lifecycle of an activity is defined by the following Activity methods. All of these are hooks that you can override to do appropriate work when the activity changes state. All activities will implement onCreate(Bundle) to do their initial setup; many will also implement onPause() to commit changes to data and otherwise prepare to stop interacting with the user. You should always call up to your superclass when implementing these methods.

- *onCreate()*: Called when the activity is first created. This is where you should do all of your normal static set up: create views, bind data to lists, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one. Always followed by onStart().

- *onRestart()*: Called after the activity has been stopped, prior to it being started again. Always followed by onStart().
- *onStart()*: Called when the activity is becoming visible to the user. Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.
- *onResume()*: Called when the activity will start interacting with the user. At this point your activity is at the top of the activity stack, with user input going to it. Always followed by onPause().
- *onPause ()*: Called as part of the activity lifecycle when an activity is going into the background, but has not (yet) been killed. The counterpart to onResume(). When activity B is launched in front of activity A, this callback will be invoked on A. B will not be created until A's onPause() returns, so be sure to not do anything lengthy here.
- *onStop()*: Called when you are no longer visible to the user. You will next receive either onRestart(), onDestroy(), or nothing, depending on later user activity. Note that this method may never be called, in low memory situations where the system does not have enough memory to keep your activity's process running after its onPause() method is called.
- *onDestroy()*: The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called finish() on it, or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the isFinishing() method. [7]

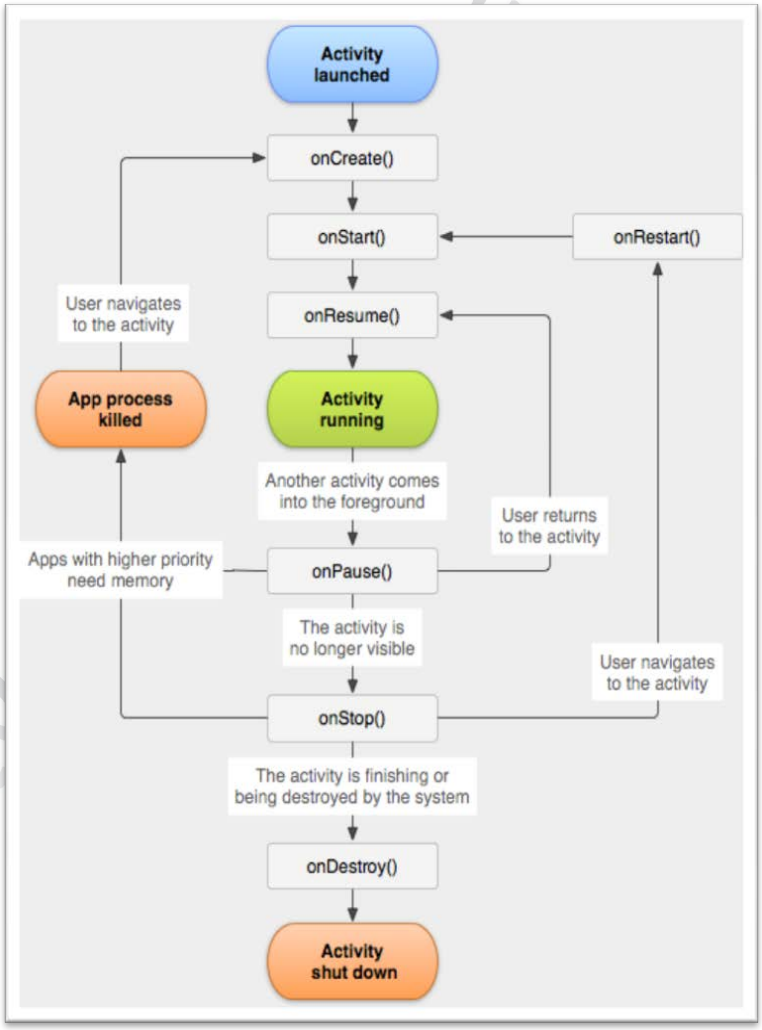

Figure 3: Activity Lifecycle [8]

1.4.2 Intents, Intent filters and receivers

Unlike ContentProviders, the other three component types of an application (activities, broadcast receivers and services) are activated through intents. An Intent is an asynchronously sent message object including the message that should be transported. The contained message either holds the name of the action that should be performed, or the name of the action being announced. The former applies to activities and services, the latter to broadcast receivers.

The Intent class has some actions like ACTION EDIT and ACTION VIEW or for broadcasts ACTION TIME TICK included already. In addition to the action, the message contains a Uniform Resource Identifier (URI) that specifies the data used for the given action. Optionally the Intent object can hold a category, a type, a component name, extra data and flags.

Android utilizes different hooks in the application components to deliver the intents. For an activity, it's onNewIntent() method is called, at a service the onBind() method is called. Broadcast actions can be announced using Context.sendBroadcast() or similar methods. Android sends the Intent to the onReceive() method of all matching registered receivers. Intents can be filtered by an application to specify which intents can be processed by the application's components. The list of filters is set in the application's manifest file, thus Android can determine the allowed intents before starting an application. [7]

1.4.3 Content Provider

The data storage and retrieval in Android applications is done via content providers. These providers can also be used to share data between multiple applications, given that the involved applications own the correct permissions to access the data. Android already has default providers for e.g. images, videos, contacts and settings which can be found in theandroid.provider package. An application queries a ContentResolver which returns the appropriate ContentProvider. All providers are accessed like databases with a URI to determine the provider, a field name and the data type of this field. Applications only access content providers via a ContentResolver, never directly. If an application wants to store data that does not have to be shared, it can use a local SQLiteDataBase. [7]

1.5 RPC

Android has a Common Object Request Broker Architecture (CORBA) and Component Object Model (COM) like lightweight RPC mechanism and brings its own language, AIDL (Android Interface Definition Language). AIDL uses proxy classes to pass messages between the server and the client. The aidl tool creates Java interfaces from the interface definition which have to be available at the client as well as at the server.

The created interface has an abstract inner class called Stub which has to be extended and implemented by the client and the server like shown in figure 7. If the server provides a Service, it has to return an instance of the interface implementation at its onBind() method. Only methods are allowed in AIDL and all of them are synchronous. [7]

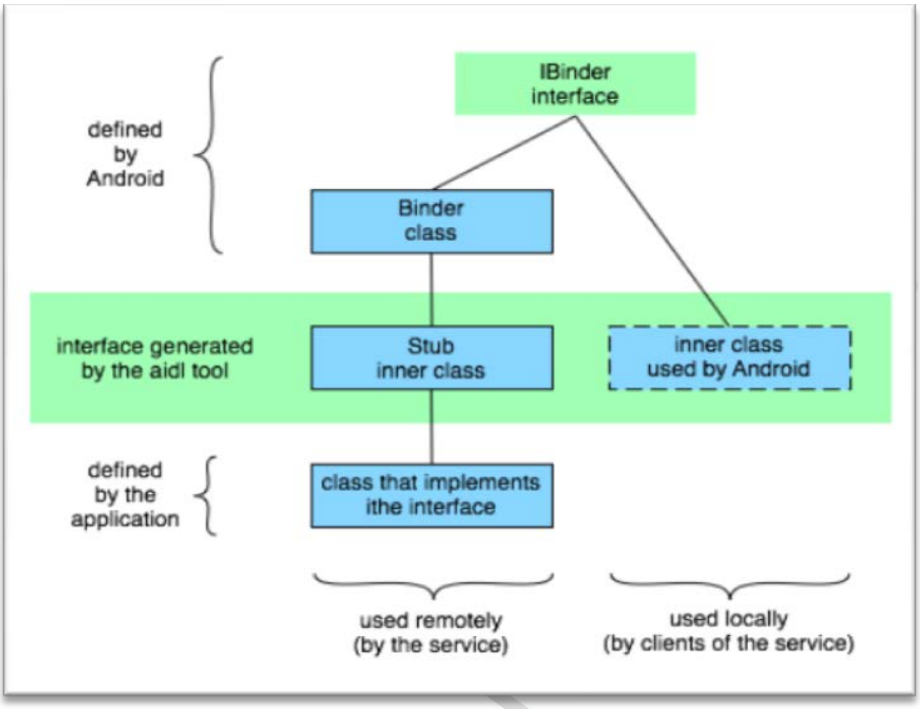

Figure 4: Android RPC Class Diagram [7]

1.6 Application Security

The security model of Android heavily depends on the multi-user capabilities of the underlying Linux. Each application runs with its own unique user id and in its own process. All Dalvik applications run in a sandbox that by default prohibits e.g. communication with other processes, access to others data, access to hardware features like GPS or camera and network access.

Opposing to platforms with native binary executables, Android makes it easy to enforce a certain application behavior, as its application VM Dalvik directly controls code execution and resource access. Platforms like iOS, webOS, Symbian or MeeGo do not have this opportunity, thus their sandboxing systems are based on means on kernel- file system- or process-level and some of these means are used by Android too.

The basic sandbox denies all requests from an application unless the permissions are explicitly granted. The permissions are set up in the application manifest file with the <uses-permission> tag. That allows the system to ask the user or a package manager upfront at install time for the application's wanted permissions.

Once installed, an application can assume that all wanted permissions are granted. During the installation process, an application is assigned with a unique and permanent user id. This user id is used to enforce permissions on process and file system level [7]. This allows the kernel to keep apps confined in memory, restrict access to underlying hardware or services, and restrict access to the file system on the device. By default, each app's data and resources are contained in a location which only the app and the core framework can access. This design is central to the Android security model. [12]

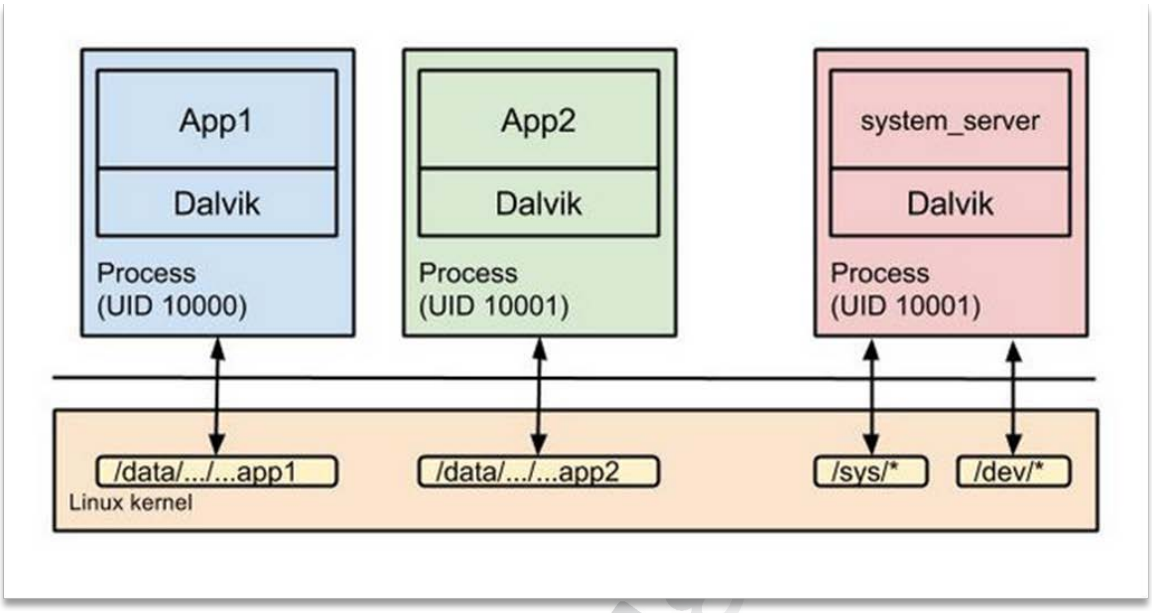

Figure 5: Android Security via the Android App Process Sandbox [11]

An application can explicitly share files by setting the file mode to be world readable or writeable, otherwise all files are private. If two applications should share the same processes or use the same user id, both of the applications have to be signed with the same certificate and ask for the same sharedUserId in their manifest files. The individual components of an application can also be restricted, to ensure that only certain sources are allowed to start an activity, a service or send a broadcast to the application's receiver. A service can enforce fine grained permissions with the Context.checkCallingPermission() method. Content providers can restrict overall read and write access as well as grant permissions on an URI basis. [7]

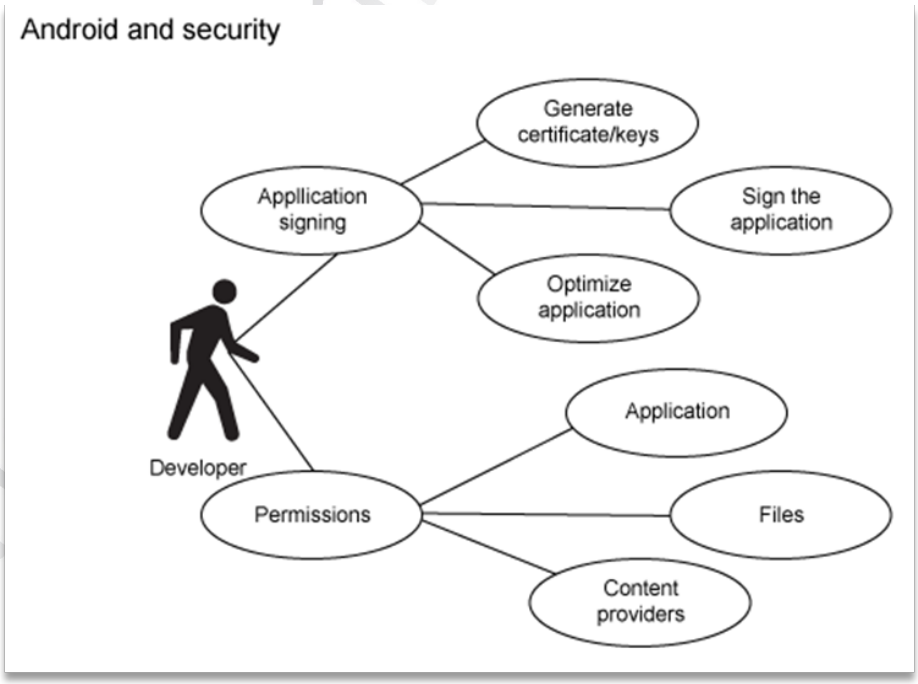

Figure 6: Android Application Security

1.7 Native Applications

Android applications that need more performance than the Dalvik VM can offer, can be partitioned. One part stays in the Dalvik VM to provide the application UI and some logic and the other part runs as native code. This way, applications can take advantage of the device capabilities, even if Android or Dalvik do not offer a certain feature.

The native code parts of an application are shared libraries which are called through the Java Native Interface (JNI). The shared library has to be included in the applications .apk file and explicitly loaded. The code from the native library is loaded into the address space of the application's VM. This leads to a possible security hole, as the security means described in section 1.6 do not cover native code.

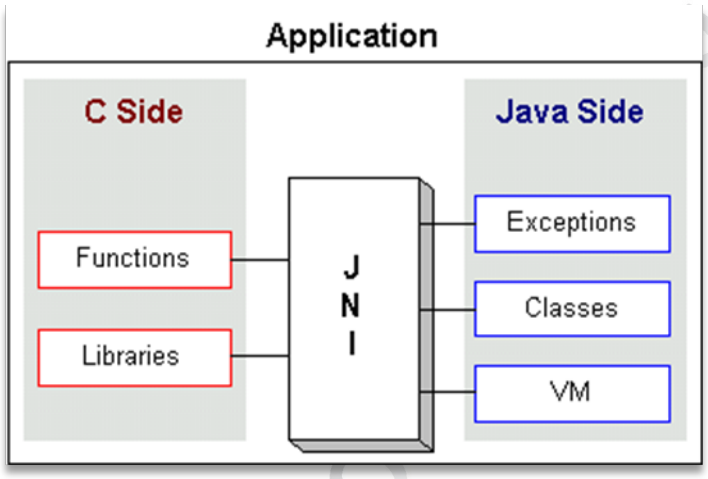

Figure 7: JNI Overview [14]

As of revision 4 of the Native Development Kit (NDK), the code can make use of the ARMv5TE and ARMv7-A instructions sets. In addition the Vector Floating Point (VFP) and Neon (Single Instruction Multiple Data instructions) extensions can be used in ARMv7-A code. Code used by native applications should only make use of a limited library set that amongst others includes the libc, libm, libz and some 2D and 3D graphics libraries.

According to performance speedup of native code over Dalvik interpreted code examinations [14], there is a huge difference between native and interpreted code, but the benchmarks were run on an early Android version that e.g. did not offer a just in time compiler [9].

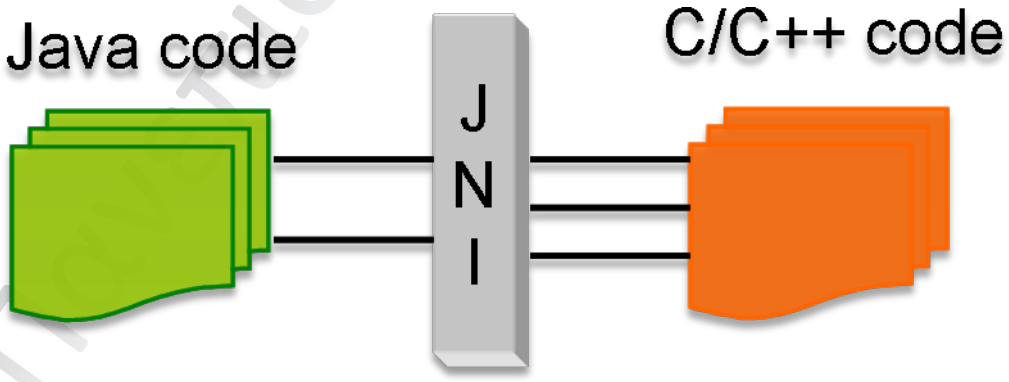

Figure 8: Native Method Java [13]

1.8 Android Architecture

Android operating system is a stack of software components which are roughly divided into five sections and four main layers as shown in the architecture diagram.

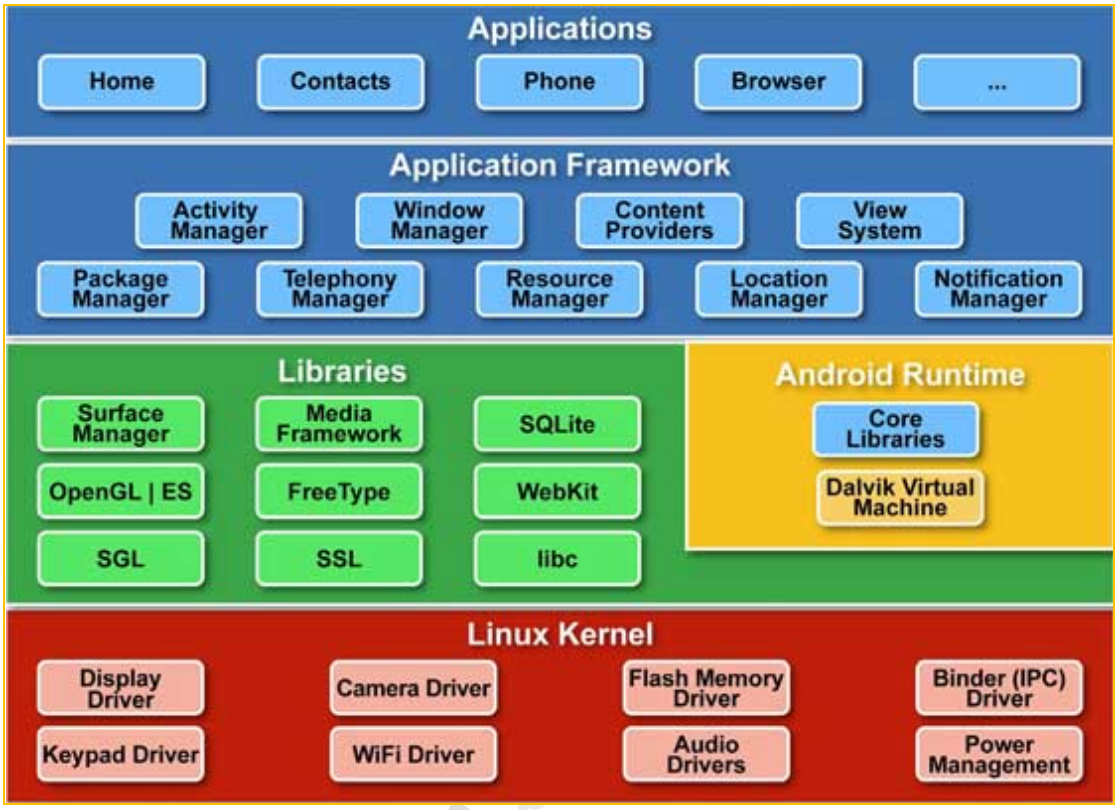

Figure 9: Android System Architecture

1.8.1 Linux Kernel

At the bottom of the layers is Linux - Linux 2.6 with approximately 115 patches. This provides basic system functionality like process management, memory management, device management like camera, keypad, display etc. Also, the kernel handles all the things that Linux is really good at such as networking and a vast array of device drivers, which take the pain out of interfacing to peripheral hardware. [15]

What is more, the kernel used in Android is modified to fulfill some special needs of the platform. The kernel is mostly extended by drivers, power management facilities and adjustments to the limited capabilities of Android's target platforms. The power management capabilities are crucial on mobile devices, thus the most important changes can be found in this area.

Like the rest of Android, the kernel is freely available and the development process is visible through the public Android source repository. There are multiple kernels available in the repository like an architecture unspecific common kernel and an experimental kernel. Some hardware specific kernels for platforms like MSM7xxx, Open Multimedia Application Platform (OMAP) and Tegra exist in the repository too.

The changes to the mainline kernel can be categorized into: bug fixes, facilities to enhance user space (lowmemorykiller, binder, ashmem, logger, etc.), new infrastructure (esp. wake locks) and support for new SoCs (msm7k, msm8k, etc.) and boards/devices.

The Android specific kernels and the mainline Linux kernel are supposed to get merged in the future, but this process is slow and will take some time.

As far Wake Locks are concerned, Android allows user space applications and therefore applications running in the Dalvik VM to prevent the system from entering a sleep or suspend state. This is important

because by default, Android tries to put the system into a sleep or better a suspend mode as soon as possible.

Applications can assure e.g. that the screen stays on or the CPU stays awake to react quickly to interrupts. The means Android provides for this task are wake locks. Wake locks can be obtained by kernel components or by user space processes. The user space interface to create a wake lock is the file /sys/power/wake lock in which the name of the new wake lock is written. To release a wake lock, the holding process writes the name in /sys/power/wake unlock. The wake lock can be furnished with a timeout to specify the time until the wake lock will be released automatically. All by the system currently used wake locks are listed in /proc/wake locks.

The kernel interface for wake locks allows to specify whether the wake lock should prevent low power states or system suspend. A wake lock is created by wake lock init() and deleted by wake lock destroy(). The created wake lock can be acquired with wake lock() and released with wake unlock(). Like in user space it is possible to define a timeout for a wake lock.

The concept of wake locks is deeply integrated into Android as drivers and many applications make heavy use of them. This is a huge stumbling block for the Android kernel code to get merged into the Linux mainline code. [7]

1.8.2 Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc. [16]

1.8.3 Android Runtime

This is the third section of the architecture and available on the second layer from the bottom. This section provides a key component called Dalvik Virtual Machine which is a kind of Java Virtual Machine specially designed and optimized for Android. The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.

The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language. [15]

Android applications and the underlying frameworks are almost entirely written in Java. Instead of using a standard Java virtual machine, Android uses its own VM. This virtual machine is not compatible to the standard Java virtual machine Java ME as it is specialized and optimized for small systems. These small systems usually only provide little RAM, a slow CPU and other than most PCs no swap space to compensate the small amount of memory.

At this point Android tells itself apart from other mobile operating systems like Symbian, Apple's iOS or Palm's webOS which use native compiled application code. The main programming languages used there are C, C++ and objective C, whereat e.g. webOS allows other mostly web based languages like JavaScript and HTML as well. The necessary byte code interpreter – the virtual machine – is called Dalvik.

Instead of using standard byte code, Dalvik has its own byte code format which is adjusted to the needs of Android target devices. The byte code is more compact than usual Java byte code and the generated .dex files are small.

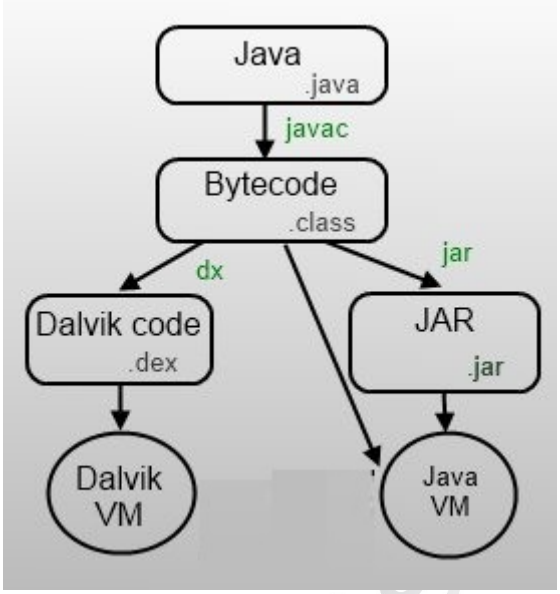

Figure 10: Dalvik Virtual Machine of Android

As a multitasking operating system, Android allows every application to be multithreaded and also to be spread over multiple processes. For the sake of improved stability and enhanced security each application is separated from other running applications. Every application runs in a sandboxed environment in its own Dalvik virtual machine instance. This requires Dalvik to be small and only add little overhead.

Dalvik is designed to run on devices with a minimum total memory of just 64 MB of RAM. For better performance actual devices have more than 64 MB installed. Of these 64 MB only about 40 MB remain for applications, libraries and services. The used libraries are quite large and likely need 10 MB of RAM. Actual applications only have around 20 MB left of the 64 MB of RAM. This very limited amount of memory has to be used efficiently in order to run multiple applications at once.

There are two major areas to consider for minimizing the memory usage. Firstly the application itself has to be as small as possible and secondly the memory allocation of each application has to be optimized. In addition to the reduced usage of valuable memory one gains faster application load times and less needed disk storage. Most importantly the power supply by battery and the used CPUs confine the allowed and possible overhead for Dalvik. The typically used ARM CPUs only provide fairly limited computational power and small caches.

Java applications for Dalvik get compiled like other Java programs with the same compilers and mostly the same toolchain. Instead of compressing and packaging the resulting class files into a .jar file, they are translated into .dex files by the dx tool. These files include the Dalvik byte code of all Java classes of the application.

Together with other resources like images, sound files or libraries the .dex files are packaged into .apk files. In order to save storage space, .dex files only contain unique data. If multiple class files share the same string, this string would only exist once in the .dex file and the multiple occurrences are just pointers to this one string (see figure 14). The same mechanism is used for method names, constants and objects which results in smaller files with much internal “pointing”. The results of these means in terms of file size can be seen in table 1.

	System libraries	**Browser app**	**Alarm clock app**
Uncompressed	21445320 – 100%	470312 – 100%	119200 – 100%
Compressed Jar	10662048 – 50%	232065 – 49%	61658 – 52%

Dex file	10311972 – 48%	209248 – 44%	53020 – 44%

Table 1: Examples for file size reduction of .dex files. The file size is given in bytes and the .dex files are not compressed [7].

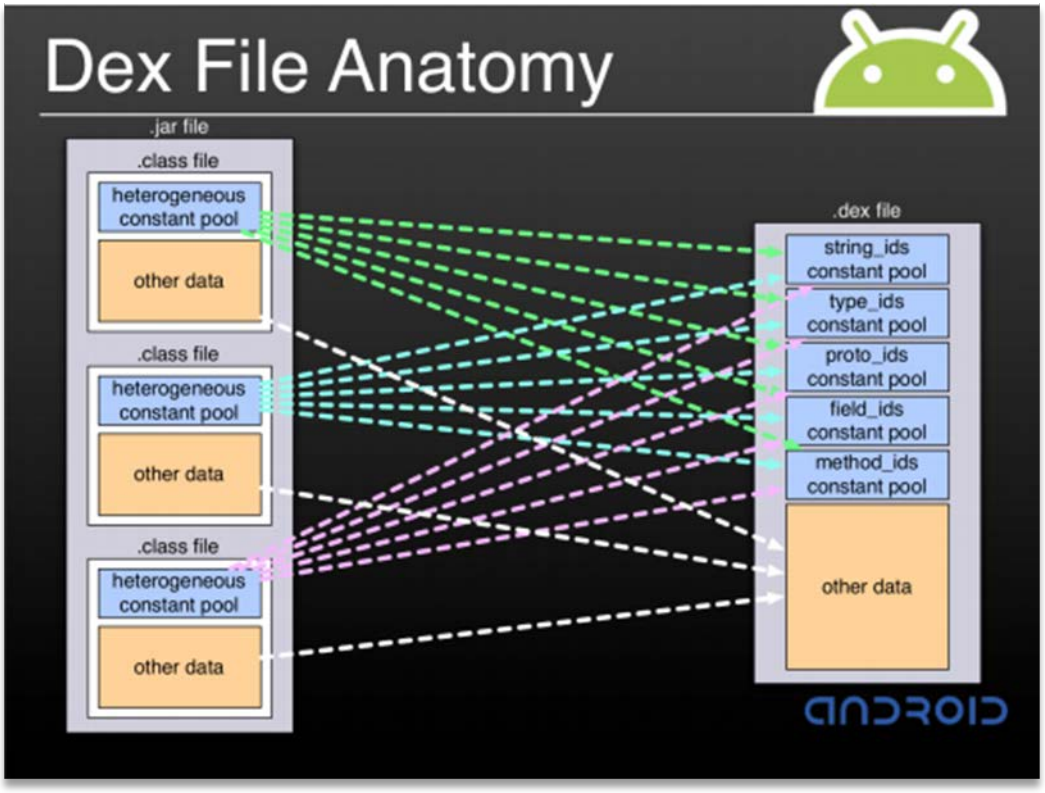

Figure 11: Dex File Anatomy [7]

Regarding the ***Dalvik byte code***, it is designed to reduce the number of necessary memory reads and writes and increased code density compared to Java byte code. For this purpose Dalvik uses its own instruction set with a fixed instruction length of 16 bit. Furthermore Dalvik is a register based virtual machine which among other things reduces the needed code units and instructions. The given register width is 32 bit which allows each register to hold two instructions, 64 bit values are hold in adjacent registers. Instructions shorter than 16 bit zero-fill the unused bits and pad to 16 bit.

Dalvik knows 256 different op codes whereof 23 are unused in Android 2.2, leading to an actual total op code number of 233. Optimizations of the byte code is mostly done by the dexopt tool at installation time.

As for the start of an application, the associated .dex file has at least to be verified for structural integrity. This verification can take some time and is needed only once, as application files don't change unless the application is updated or uninstalled. Due to the specific fact, the verification process can be done either at the first startup or during the installation or update process. In Android this verification process is done at installation time, so that at all application startups later only e.g. a checksum of the .dex file is needed to verify it.

The verification and optimization procedure is performed by the dexopt program. To verify a .dex file, dexopt loads all classes of the file into a briefly initialized VM and runs through all instructions in all methods of each class. This way illegal instructions or instruction combinations can be found before the application actually runs for the first time. Classes in which the verification process succeeded get marked by a flag to avoid rechecking this class again. In order to check the .dex files integrity, a CRC-32 checksum is stored within the file.

After the successful verification of a .dex file, it gets optimized by dexopt. The optimizations aim at performance increase through reduced code size or reduced code complexity. The optimization mechanisms heavily depend on the target VM version and the host platform which makes it hard to run dexopt elsewhere than on the host. The resulting code is unoptimized Dalvik byte code mixed with optimized code using op codes not defined in the Dalvik byte code specification. If necessary for the processor architecture endianness, the code is byte swapped and aligned accordingly.

For code reduction dexopt prunes all empty methods and replaces them with a no-operation op code. Inlining some very often called methods like String.length() reduces the method call overhead and virtual method call indices get replaced by vtable indices. Furthermore field indices get replaced by byte offsets and short data types like char, byte and Boolean are merged into 32 bit form to save valuable CPU cache space. If it is possible to pre-compute .dex file data, dexopt appends the resulting data which reduces the CPU time needed by the executing VM later.

Both verification and optimization are limited to process only one .dex file at a time which leads to problems at handling dependencies. If a .dex file is optimized, it contains a list of dependencies which can be found in the bootstrap class path. In order to guarantee consistency in case of exchanged .dex files, only dependencies to .dex files in the bootstrap class path are allowed. This way the verification of methods depending on external .dex files other than those in bootstrap will fail and the related class will not be optimized. [7]

In Dalvik there are four different kinds of memory to distinguish that can be grouped to clean/dirty and shared/private. Typical data residing in either shared or private clean memory are libraries and application specific files like .dex files. Clean memory is backed up by files or other sources and can be pruned by the kernel without data loss. The private dirty memory usually consists of the applications heap and writeable control data structures like those needed in .dex files. These three categories of different memory are quite common and no specialty of Dalvik.

Shared dirty memory is possible through a facility of Dalvik called ***Zygote***. It is a process which starts at boot time and is the parent of all Dalvik VMs in the system. The Zygote loads and initializes classes that are supposed to be used very often by applications into its heap. In shared dirty memory resides e.g. the dex data structures of libraries.

After the startup of the Zygote, it listens to commands on a socket. If a new application starts, a command is sent to the Zygote which performs a standard fork(). The newly forked process becomes a full Dalvik VM running the started application. The shared dirty memory is “copy-on-write” memory to minimize the memory consumption. [7]

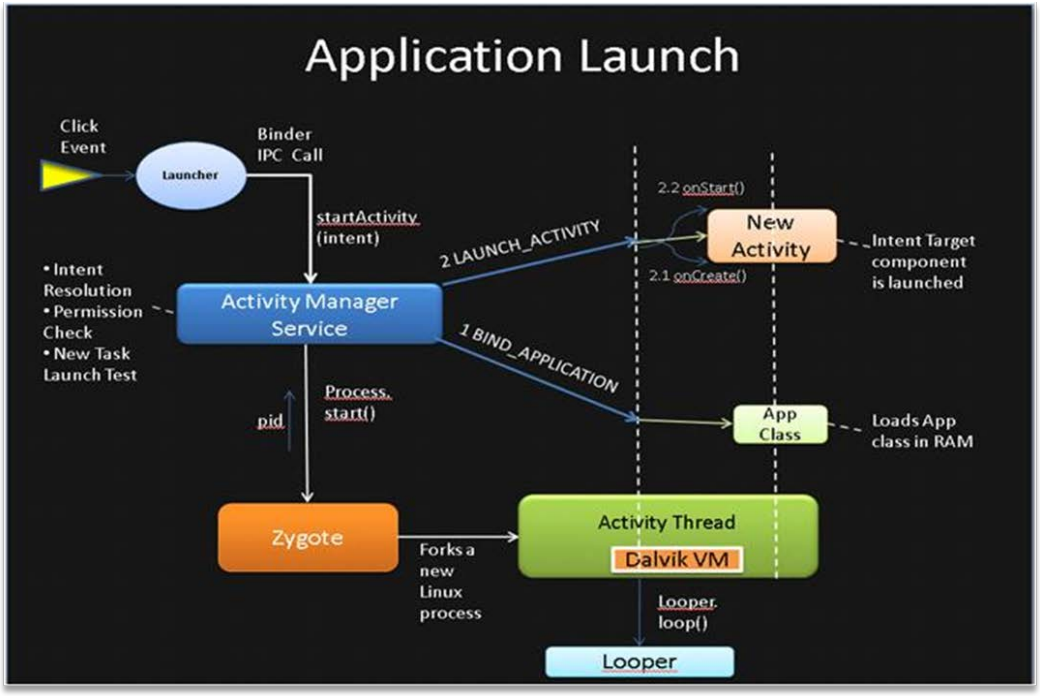

Figure 12: Application Launch [18]

Another significant of Dalvik is the ***Garbage Collector (GC)***. Garbage Collector in Android runs in each VM separately, therefore each VMs heap is garbage collected independently. The Zygote process and the concept of shared dirty memory requires the GC data structures ("mark bits") to not be tied to the objects on the heap, but kept separate. If the mark bits would lie next to the objects on the heap, a run of the GC would touch these bits and turn the shared dirty memory into private dirty memory. In order to minimize the private dirty memory the needed mark bits are allocated just before a GC run and freed afterwards. [8]

1.8.4 Application Framework

The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications [15].

The application framework of Android provides APIs in various areas like networking, multimedia, graphical user interface, power management and storage access. The libraries in the framework are written in Java and run on top of the core libraries of the Android Runtime. These core libraries utilize and encapsulate optimized native system libraries like libc, libssl and FreeType.

The Application Framework provides managers for different purposes like power management, resource handling, system wide notification and window management. Applications are supposed to use the services of the managers and not use the underlying libraries directly. This way it is possible for the managers to enforce application permissions through the means of the sandbox permission system. The managers can ensure that an application is allowed to e.g. initiate a phone call or send data over the network.

As the development of Android continues, new ***APIs*** are added and old APIs get marked obsolete and are eventually erased. Each Android version has its own API level and applications can define a minimum, maximum and preferred API level in their manifest file. The API level changes between major as well as minor releases of Android, as shown in table 2. [7]

Platform Version	API Level
Android 2.2	8
Android 2.1	7
Android 2.0.1	6
Android 2.0	5
Android 1.6	4
Android 1.5	3
Android 1.1	2
Android 1.0	1

Table 2: The API levels of all Android versions [7]

The basis of a graphical ***user interface*** in Android is the View class. All visible elements of a user interface and some invisible items are derived from this class. Android provides standardized UI elements like buttons, text and video views or a date picker. View items can be partitioned to a ViewGroup which allows applying a layout to the views in the group.

In order to interact with the UI, the View class provides hook methods like onClick(), onTouch() and onKey() which are called by the underlying framework. UI events can also be received by listeners that provide the onXYZ() method and are registered like e.g. the OnKeyListener.

In addition to the described UI elements, the framework provides menus and dialogs. Menus can either be options menus that can be accessed via the menu key, or they are context menus of a view. To inform the user or to obtain input, an application can present a dialog. Android provides dialogs for date and time picking, progress notification and a customizable general dialog with buttons.

For user notification exist three different mechanisms in Android. The most intrusive one is the notification with a dialog which moves the focus to the dialog, leaving the application in the background. The toast notification displays a text on top of the current application that allows no interaction and disappears after a given timeout. The status bar notification leaves the current application appearance unaffected and puts an icon in the status bar. The toast notification as well as the status bar notification can be utilized by background services. [7]

The ***media framework*** supports multiple audio, video and image data formats and has a media player and an encoder built into Android. The number of data formats can be extended, but as a minimum the following decodeable formats are supported:

- **Image** JPEG (encoder provided), PNG, GIF and BMP
- **Audio** MP3, OGG Vorbis, MIDI, PCM/WAVE, AAC, AAC+ and AMR (encoder provided)
- **Video** H.263 (encoder provided), H.264 and MPEG-4

The MediaPlayer and MediaRecorder classes can be used to play back and record the supported multimedia data. In addition to the player and recorder, the android.media package includes specialized classes to play alarms and ring tones or generate image thumbnails and set up the camera. [8]

Access to ***networks*** and especially the Internet is crucial for Android devices, as many applications depend on network access. Most Android devices have multiple technologies on board to gain network access. Smartphones at least have Internet access via GPRS (2G telephony) or UMTS and usually Wi-Fi is available too.

The ConnectivityManager allows applications to check the status of the different access technologies and it informs applications via broadcasted intents about connectivity changes. To make use of the network connection, Android provides a broad range of packages and classes.

- **java.net.*** The standard Java network classes like sockets, simple HTTP and plain packets
- **android.net.*** Extended java.net capabilities
- **android.net.http.*** SSL certificate handling

- **org.apache.*** Specialized HTTP
- **android.telephony.*** GSM & CDMA specific classes, send text message, signal strength and status information
- **android.net.wifi** Wi-Fi configuration and status. [7]

Android's ***Bluetooth*** APIs allows applications to search for other devices, pair with them and exchange data. In order to use the Bluetooth capabilities, an application needs to have the BLUETOOTH or BLUETOOTH ADMIN permission in its manifest file granted. The Bluetooth API is located in the android.bluetooth package.

The connection via a RFCOMM (RS-232 serial line via Bluetooth) compatible BluetoothSocket is initiated by a local BluetoothAdapter and targeted to a remote BluetoothDevice. Making the device visible for scans and pairing devices need user interaction as the permission system asks for granting the needed permissions. [7]

Applications can ***store and retrieve*** their data in various ways:

- **SharedPreferences:** A class that provides storage for key/value pairs of primitive data types. The data can be shared between multiple clients in the same process. All data can only be modified through an Editor object to ensure consistency even if the application is terminated.
- **Internal/external storage:** If an application wants to make sure that no other application or even the user can access saved data, it can write this data to the internal storage. Files saved to the internal storage can be set up to allow others to access the files. Files that are supposed to be shared and/or user visible, can be stored to the external storage. Files in this storage area are always publicly visible and accessible. Android allows the external storage to be turned into an USB mass storage with full access to all files in this storage area.
- **Database storage:** The SQLite database files behind all content providers are available to applications as an application private database back end.
- **Network storage:** With network access available, Android applications can obtain and store their data via sockets, HTTP connections and other means from java.net.* and android.net.*.
- **Application data backup:** From version 2.2 on Android provides a mechanism to backup preferences and application data on a remote site – the cloud. Applications can request a backup and Android automatically restores the data on an application reinstall, if the application is installed on a new device or by request of the application. The mechanism consists of three parts on the device: *the backup agent, transport and manager.*
  Applications can announce a backup agent in the manifest file and implement it to provide hooks for the backup manager. If application data changes, the application can inform the backup manager which may schedule a backup and call the according backup agent hook. The backup mechanism does not allow on demand read and write access to the backed up data as it's always the backup manager's decision when to perform a restore or a backup.
  The backup transport is responsible to transfer the data from or to the remote site. There is no guarantee that the backup capability is available on a device as the backup transport depends on the device manufacturer and the service provider. [7]

Supplementary to the already mentioned APIs, Android provides a wide range of ***other APIs*** for many different areas. Some of those packages are listed below:

- **Location & Maps:** Applications can use the LocationManager and classes from android.location to obtain the device's location directly on demand or by a broadcast. The external com.google.android.maps package provides mapping facilities for applications.
- **Search:** The system wide search on Android devices includes not only contacts, web search and such, but can also be extended by applications to make their content provider data searchable. The SearchManager provides unified dialogs and extra functionality like voice search for all applications.
- **WebKit:** For browsing web pages, Android provides among other things a WebKit based HTML renderer, a JavaScript engine (V8) and a cookie manager. These facilities can be used by applications to provide browser functionality inside the application.

- **Speech:** The android.speech package puts applications in the position to make use of server side speech recognition services. A text-to-speech API provides the means to turn text into audio files or play back the result directly.
- **C2DM:** Cloud to Device Messaging –A new and in Android 2.2 fairly limited capability to send data like URLs or even intents to the device. [7]

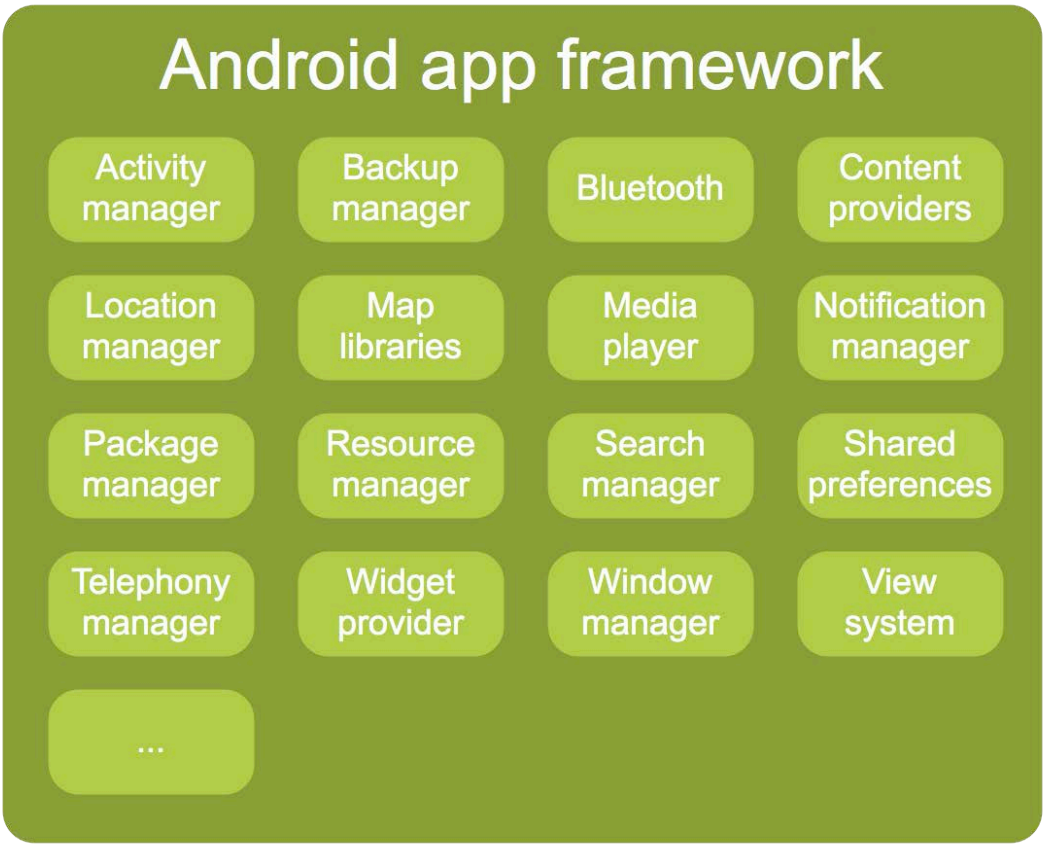

Figure 13: Android Application Framework [18]

1.8.5 Applications

You will find all the Android application at the top layer. You will write your application to be installed on this layer only. Examples of such applications are Contacts Books, Browser, and Games. [15]

2. Cryptographic Principles

2.1 Introduction to Cryptography

The word cryptography comes from the Greek words “κρυπτό” (i.e. secret) and “γραφή” (i.e. writing). Cryptography is the art of secret writing. The basic service provided by cryptography is the ability to send information between participants in a manner that prevents others from reading it.

A message in its original form is known as plaintext or cleartext. The information in which cryptography is applied is known as ciphertext. The process for producing ciphertext from plaintext is known as encryption. The reverse of encryption is called decryption. While cryptographers invent clever secret codes (also called ciphers), cryptanalysts attempt to break these codes.

Figure 14: Encryption and Decryption Processes [20]

These two disciplines constantly try to keep ahead of each other. Ultimately, the success of the cryptographers rests on the fundamental tenet of Cryptography: *“If lots of smart people have failed to solve a problem, then it probably won’t be solved (soon)”*.

Cryptographic systems tend to involve both an algorithm and a secret value. The secret value is known as the key. The reason for having a key in addition to an algorithm is that it is difficult to keep devising new algorithms that will allow reversible scrambling of information, and it is difficult to quickly explain a newly devised algorithm to the person with whom you’d like to start communicating securely. With a good cryptographic scheme it is an accepted practice to have everyone, including the adversaries know the algorithm while knowledge of the algorithm without the key does not help decrypt the information.

The concept of a key is analogous to the combination for a combination lock. Although the concept of a combination lock is well known, you can’t open a combination lock easily without knowing the combination. [20]

2.2 Cryptographic Functions & Algorithms

There are three types of cryptographic functions: *hash functions, private key functions, and public key functions*. The basic difference between the three methods of encryption is that Public Key Cryptography involves the use of two keys, Private Key Cryptography involves the use of one key and Hash functions do not involve the use of any key. Nevertheless, there is yet another classification of cryptographic algorithms, to *block ciphers* and *stream ciphers*, which describe the manner that the cryptographic algorithm processes the data. In the following paragraphs all of the aforementioned types of encryption are described in full detail, beginning with the latter classification. [20]

2.2.1 Block Ciphers

Block ciphers are the central tool in the design of protocols for symmetric cryptography. They are the main available “technology" we have at our disposal. In this chapter we will elaborate on these objects and describe their construction.

A block cipher is a deterministic algorithm operating on fixed-length groups of bits, called blocks, with an unvarying transformation that is specified by a symmetric key and it is of utmost importance to understand that block ciphers are just tools. Block ciphers don't, by themselves, do something that an end-user would care about. As with any powerful tool, one has to learn to use this one. Even an excellent

block cipher won't give the cryptographers security if they don't use them properly, in a right manner. But used well, these are powerful tools indeed.

More specifically, a block cipher is a function E: $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$. This notation means that *E* takes two inputs, one being a k-bit string and the other an n-bit string, and returns an n-bit string. The first input is the key. The second might be called the plaintext, and the output might be called a ciphertext. The key-length *k* and the block-length *n* are parameters associated to the block cipher. They vary from block cipher to block cipher, as of course does the design of the algorithm itself.

For each key $K \in \{0, 1\}^k$ we let E_K: $\{0, 1\}^n \rightarrow \{0, 1\}^n$ to be the function defined by $E_K(M) = E(K, M)$. For any block cipher, and any key *K*, it is required that the function E_K be a permutation on $\{0, 1\}^n$. This means that it is a bijection (i.e. a one-to-one and on to function) of $\{0, 1\}^n$ to $\{0, 1\}^n$. (For every $C \in \{0, 1\}^n$ there is exactly one $M \in \{0, 1\}^n$ such that $E_K(M)=C$). Accordingly E_K has an inverse, and we denote it E^{-1}_K. This function also maps $\{0, 1\}^n$ to $\{0, 1\}^n$, and of course we have $E^{-1}_K(E_K(M)) = M$ and $E_K(E^{-1}_K(C)) = C$ for all $M,C \in \{0, 1\}^n$. We let E^{-1}: $\{0, 1\}^k \times \{0, 1\}^n \rightarrow \{0, 1\}^n$ be defined by $E^{-1}(K,C)=E^{-1}_K(C)$. This is the inverse block cipher to *E*.

The block cipher *E* is a public and fully specified algorithm. Both the cipher *E* and its inverse E^{-1} should be easily computable, meaning given *K, M* we can readily compute *E(K,M),* and given *K,C* we can readily compute $E^{-1}(K,C)$. By “readily compute" we mean that there are public and relatively efficient programs available for these tasks.

In typical usage, a random key *K* is chosen and kept secret between a pair of users. The function E_K is then used by the two parties to process data in some way before they send it to each other. Typically, we will assume the adversary will be able to obtain some input-output examples for E_K, meaning pairs of the form *(M,C)* where $C=E_K(M)$. But, ordinarily, the adversary will not be shown the key *K*. Security relies on the secrecy of the key. The block cipher should be designed to make this task computationally difficult. [22]

2.2.2 Stream Ciphers

A stream cipher is a symmetric cipher which operates with a time-varying transformation on individual plaintext digits. By contrast, block ciphers operate with a fixed transformation on large blocks of plaintext digits. More precisely, in a stream cipher a sequence of plaintext digits, $m_0m_1...$, is encrypted into a sequence of ciphertext digits $c_0c_1...$ as follows: a pseudo-random sequence $s_0s_1...$, called the running-key or the keystream, is produced by a finite state automaton whose initial state is determined by a secret key. The *i-th* keystream digit only depends on the secret key and on the $(i-1)$ previous plaintext digits. Then, the *i-th* ciphertext digit is obtained by combining the *i-th* plaintext digit with the *i-th* keystream digit.

Stream ciphers are classified into two types: synchronous stream ciphers and asynchronous stream ciphers. In addition, stream ciphers have several advantages which make them suitable for some applications. Most notably, they are usually faster and have a lower hardware complexity than block ciphers. They are also appropriate when buffering is limited, since the digits are individually encrypted and decrypted. Moreover, synchronous stream ciphers are not affected by error-propagation. [23]

2.2.3 Private Key Cryptography

Private Key Cryptography, also known as Secret Key Cryptography or Symmetric Encryption, involves two parties who share a single key to imbue communicated data with various security attributes. Given a message (plaintext) and the key, encryption produces unintelligible data (ciphertext), which is about the same length as the plaintext was. Decryption is the reverse of encryption, and uses the same key as encryption. The main security goals are privacy and authenticity of the communicated data. [20][21]

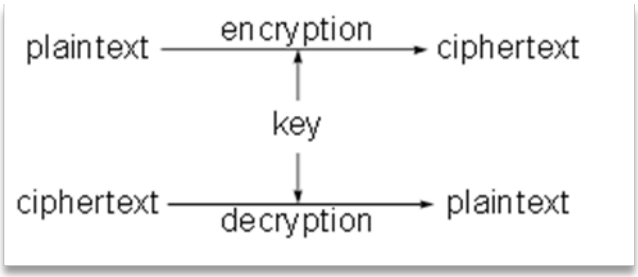

Figure 15: Encryption and Decryption in Private Key Cryptography [20]

In Private Key Cryptography, cryptographers employ a variety of symmetric encryption schemes. Such a scheme specifies an encryption algorithm, which tells the sender how to process the plaintext using the key, thereby producing the ciphertext that is actually transmitted. An encryption scheme also specifies a decryption algorithm, which tells the receiver how to retrieve the original plaintext from the transmission while possibly performing some verification, too. Finally, there is a key-generation algorithm, which produces a key that the parties need to share. The formal description follows.

A ***symmetric encryption scheme*** *SE* = (*K*, *E*, *D*) consists of three algorithms, as follows:

•The randomized *key generation algorithm K* returns a string *K*. We let *Keys*(*SE*) denote the set of all strings that have non-zero probability of being output by *K*. The members of this set are called keys. We write $K \leftarrow K$ for the operation of executing *K* and letting denote *K* the key returned.

•The *encryption algorithm E*, which might be randomized or stateful, takes a key $K \in Keys(SE)$ and a plaintext M $\in \{0,1\}^*$ to return a ciphertext $C \in \{0,1\} \cup \{\perp\}$. We executing *E on K* and *M* and letting *C* denote the ciphertext returned.

•The deterministic *decryption algorithm D* takes a key $K \in Keys(SE)$ and a ciphertext $C \in \{0, 1\}$ to return some $M \in \{0,1\} \cup \{\perp\}$. We write $M \leftarrow D_K(C)$ for the operation of executing *D* on *K* and *C* and letting *M* denote the message returned.

The scheme is said to provide correct decryption if for any key $K \in Keys(SE)$, any se $M_1, \ldots, M_q \in \{0,1\}$, and any sequence of ciphertexts $C_1 \leftarrow E_K(M1)$, $C_2 \leftarrow E_K(M_2), \ldots, C_q \leftarrow E_K(M_q)$ that may arise in encrypting $M_1, \ldots, M_q$, it is the case that $D_K(C_i) = M_i$ for each $C_i \neq \perp$.

The key-generation algorithm, as the definition indicates, is randomized. It takes no inputs. When it is run, it flips coins internally and uses these to select a key *K*. Typically, the key is just a random string of some length, in which case this length is called the key length of the scheme. When two parties want to use the scheme, it is assumed they are in possession of a key *K* generated via *K*.

Assuming that the key has been shared, the sender, once in possession of a shared key, he can run the encryption algorithm with key *K* and input message *M* to get back a string we call the ciphertext. The latter can then be transmitted to the receiver.

The encryption algorithm may be either randomized or stateful. If randomized, it flips coins and uses those to compute its output on a given input *K*, *M*. Each time the algorithm is invoked, it flips coins anew. In particular, invoking the encryption algorithm twice on the same inputs may not yield the same response both times.

An encryption algorithm is called *stateful* when its operation depends on a quantity called the state that is initialized in some pre-specified way. When the encryption algorithm is invoked on inputs *K*, *M*, it computes a ciphertext based on *K*, *M* and the current state. It then updates the state, and the new state value is stored. (The receiver does not maintain matching state and, in particular, decryption does not require access to any global variable or call for any synchronization between parties.) Usually, when there is state to be maintained, the state is just a counter. If there is no state maintained by the encryption algorithm the encryption scheme is said to be *stateless*. The encryption algorithm might be both randomized and stateful, but in practice this is rare: it is usually one or the other but not both.

When we talk of a randomized symmetric encryption scheme we mean that the encryption algorithm is randomized. When we talk of a stateful symmetric encryption scheme we mean that the encryption algorithm is stateful.

The receiver, upon receiving a ciphertext *C*, will run the decryption algorithm with the same key used to create the ciphertext, namely compute $D_K(C)$. The decryption algorithm is neither randomized nor stateful.

Many encryption schemes restrict the set of strings that they are willing to encrypt. (For example, perhaps the algorithm can only encrypt plaintexts of length a positive multiple of some block length n , and can only encrypt plaintexts of length up to some maximum length.) These kinds of restrictions are captured by having the encryption algorithm return the special symbol ⊥ when fed a message not meeting the required restriction. In a stateless scheme, there is typically a set of strings *M*, called the plaintext space, such that $M \in \mathcal{M}$ if and only if $Pr[K \leftarrow K;\ C \leftarrow E_K(M) : C \neq \perp] = 1$.

In a stateful scheme, whether or not $E_K(M)$ returns ⊥ depends not only value of the state variable. For example, when a counter is being used, it is typical that there is a limit to the number of encryptions performed, and when the counter reaches a certain value the encryption algorithm returns ⊥ no matter what message is fed to it.

The correct decryption requirement simply says that decryption works: if a message *M* is encrypted under a key *K* to yield a ciphertext *C*, then one can recover *M* by decrypting *C* under *K*. This holds, however, only if $C \neq \perp$. The condition thus says that, for each key $K \in Keys(SE)$ and message $M \in \mathcal{M}$, with probability one over the coins of the encryption algorithm, either the latter outputs ⊥ or it outputs a ciphertext *C* which upon decryption yields *M*. If the scheme is stateful, this condition is required to hold for every value of the state.

Correct decryption is, naturally, a requirement before one can use a symmetric encryption scheme in practice, for if this condition is not met, the scheme fails to communicate information accurately. In analyzing the security of symmetric encryption schemes, however, we will see that it is sometimes useful to be able to consider ones that do not meet this condition. [22]

In the current paragraph are described specific encryption schemes. It is noted that not all of the schemes described below are secure ***encryption schemes***. Nevertheless, all the schemes satisfy the correct decryption requirement.

The **one-time-pad encryption scheme** *SE= (K, E, D)* is stateful and deterministic. The key-generation algorithm simply returns a random k-bit string *K*, where the key-length k is a parameter of the scheme, so that the key space is $Keys(SE) = \{0, 1\}^k$. The encryptor maintains a counter *ctr* which is initially zero. The encryption and decryption algorithms operate as follows:

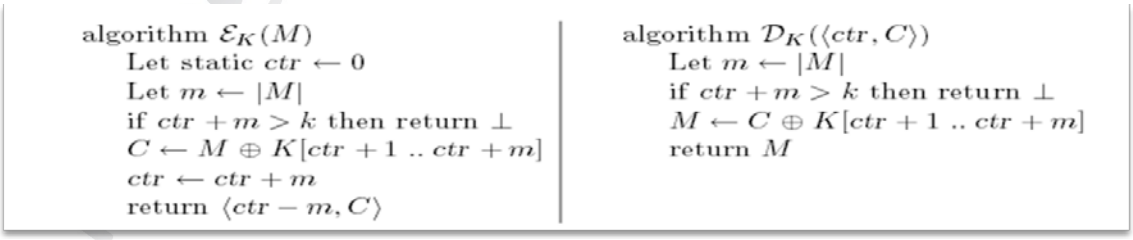

Figure 16: One-time-pad encryption and decryption [21]

Here X[i .. j] denotes the i-th through j-th bit of the binary string X. By the pair of ctr and C, we mean a string that encodes the number ctr and the string C. The most natural encoding is to encode ctr using some fixed number of bits, at least lgk, and to prepend this to C. Conventions are established so that every string Y is regarded as encoding some ctr, C for some ctr, C. The encryption algorithm XORs the message bits with key bits, starting with the key bit indicated by one plus the current counter value. The

counter is then incremented by the length of the message. Key bits are not reused, and thus if not enough key bits are available to encrypt a message, the encryption algorithm returns ⊥. Note that the ciphertext returned includes the value of the counter. This is to enable decryption. [21]

The following schemes rely either on a family of permutations (i.e., a block cipher) or a family of functions. Effectively, the mechanisms spell out how to use the block cipher to encrypt. We call such a mechanism a ***mode of operation*** of the block cipher. For these schemes it is convenient to assume that the length of the message to be encrypted is a positive multiple of a block length associated to the family. In practice, one could pad the message appropriately so that the padded message always had length a positive multiple of the block length, and apply the encryption algorithm to the padded message. The padding function should be injective and easily invertible. In this way you would create a new encryption scheme.

With a block length *n* understood, we will denote by *X[i]. . . X[m]* ← X the operation of parsing string *X* into *m*−*i*+ *1* blocks, each block of length *n*. Here i ≤ m and X is assumed to have length *(m*−*i*+ *1)n*. Thus, *X*[j] consists of bits *(j*−*i)n*+ *1* to *(j*−*i*+ *1)n* of *X*, for *i* ≤*j* ≤*m*. [21]

Let *E: K* × *{0, 1}*^{n} → *{0, 1}*^{n} be a block cipher. Operating it in ***ECB (Electronic Code Book)*** mode yields a stateless symmetric encryption scheme *SE= (K, E, D).* The key-generation algorithm simply returns a random key for the block cipher, meaning it picks a random string *K* ← *K* and returns it. Notice that the encryption algorithm does not make any random choices. (That does not mean it is not, technically, a randomized algorithm; it is simply a randomized algorithm that happened not to make any random choices). [22]

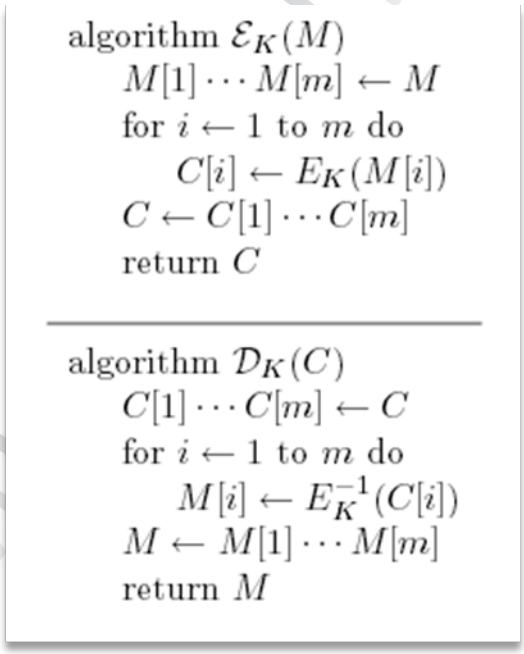

Figure 17: ECB Algorithm [21]

Let *E: K* × *{0, 1}*^{n} → *{0, 1}*^{n} be a block cipher. Operating it in ***CBC mode*** with random IV yields a stateless symmetric encryption scheme, SE= (K, E, D). The key generation algorithm simply returns a random key for the block cipher, *K* ← *K*. The IV (Initialization Vector) is *C[0]*, which is chosen at random by the encryption algorithm. This choice is made independently each time the algorithm is invoked.

For the following schemes it is useful to introduce some notation. With *n* fixed, we let *i* denote the n-bit string that is the binary representation of integer *i* mod 2^n. If we use a number I ≥ 0 in a context for which a string *I* is required, it is understood that we mean to replace I by I= $[i]_n$. [21]

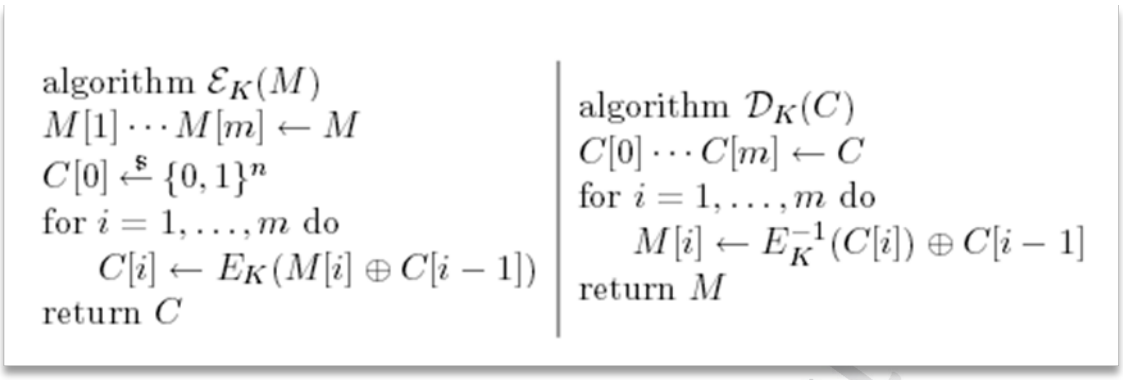

Figure 18: CBC Algorithm [21]

Let *F: K × {0, 1}^{n} → {0, 1}^{n}* be a family of functions (possibly a block cipher, but not necessarily.) Then ***CTR mode*** over *F* with a random starting point is a probabilistic, stateless symmetric encryption scheme, *SE= (K, E, D).* The key-generation algorithm simply returns a random key for *E*. The starting point *C[0]* is used to define a sequence of values on which F_K is applied to produce a "pseudo one-time pad" to which the plaintext is XORed. The starting point *C[0]* chosen by the encryption algorithm is a random n-bit string. To add an n-bit string *C[0]* to an integer I -when we write $F_K(R+i)$ - convert the n-bit string *C[0]* into an integer in the range *[0..2^n−1]* in the usual way, add this number to *i*, take the result modulo *2^n*, and then convert this back into an n-bit string. Note that the starting point *C[0]* is included in the ciphertext, to enable decryption. [22]

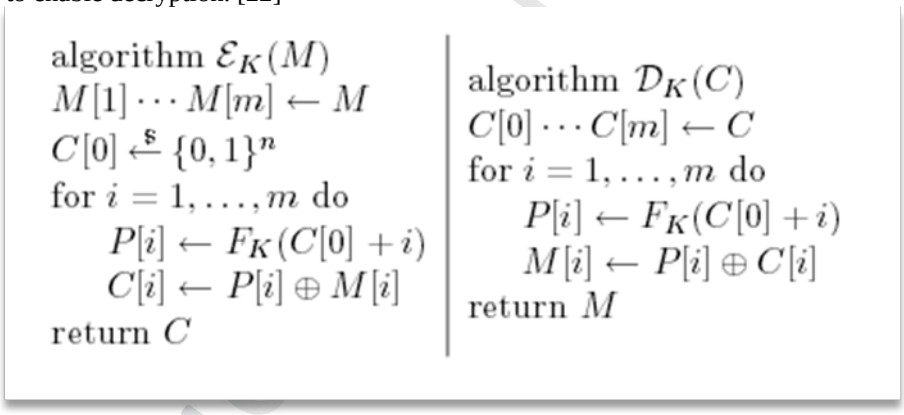

Figure 19: Randomized and Stateless algorithms for CTR [21]

2.2.4 Public Key Cryptography

Public key cryptography is sometimes also referred to as Asymmetric Cryptography. Public key cryptography is a relatively new field, invented in 1975. Unlike secret key cryptography, keys are not shared. Instead, each individual has two keys: a private key that need not be revealed to anyone, and a public key that is preferably known to the entire world.

Note that we call the private key a private key and not a secret key. This convention is an attempt to make it clear in any context whether public key cryptography or secret key cryptography is being used. It is fact that there is the tendency to use the term secret key for the private key in public key cryptography, or use the term private key for the secret key in secret key technology. The term private key must be used when referring to the key in public key cryptography that must not be made public.

In this paragraph, we will use the letter *e* to refer to the public key, since the public key is used when encrypting a message, and the letter *d* to refer to the private key, because the private key is used to decrypt a message. Encryption and decryption are two mathematical functions that are inverses of each other.

There is an additional thing one can do with public key technology, which is to generate a digital signature on a message. A digital signature is a number associated with a message, for example a checksum or the MIC (message integrity code). However, unlike a checksum, which can be generated by anyone, a digital signature can only be generated by someone knowing the private key. A public key signature differs from a secret key MIC because verification of a MIC requires knowledge of the same secret as was used to create it.

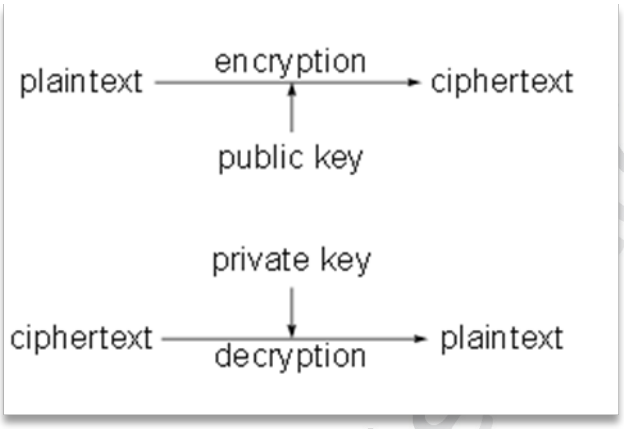

Figure 20: Asymmetric encryption and decryption processes [20]

Therefore anyone who can verify a MIC can also generate one, and so be able to substitute a different message and corresponding MIC. In contrast, verification of the signature only requires knowledge of the public key. So Alice can sign a message by generating a signature only she can generate, and other people can verify that it is Alice's signature, but cannot forge her signature. This is called a signature because it shares with handwritten signatures the property that it is possible to be able to recognize a signature as authentic without being able to forge it.

Public key cryptography can do anything secret key cryptography can do, but the known public key cryptographic algorithms are orders of magnitude slower than the best known secret key cryptographic algorithms and so are usually only used for things secret key cryptography can't do. Public key cryptography is very useful because network security based on public key technology tends to be more secure and more easily configurable. Often it is mixed with secret key technology. For example, public key cryptography might be used in the beginning of communication for authentication and to establish a temporary shared secret key, then the secret key is used to encrypt the remainder of the conversation using secret key technology.

For instance, suppose Alice wants to talk to Bob. She uses his public key to encrypt a secret key, then uses that secret key to encrypt whatever else she wants to send him. Only Bob can decrypt the secret key. He can then communicate using that secret key with whoever sent that message. Notice that given this protocol, Bob does not know that it was Alice who sent the message. This could be fixed by having Alice digitally sign the encrypted secret key using her private key. [20]

An ***asymmetric encryption scheme*** is just like a symmetric encryption scheme except for an asymmetry in the key structure. The key *pk* used to encrypt is different from the key *sk* used to decrypt. Furthermore pk is public, known to the sender and also to the adversary. So while only a receiver in possession of the secret key can decrypt, anyone in possession of the corresponding public key can encrypt data to send to this one receiver.

An asymmetric encryption scheme AE= (K, E, D) consists of three algorithms, as follows:

•The randomized key generation algorithm K (takes no inputs and) returns a pair (pk, sk) of keys, the public key and matching secret key, respectively. We write $(pk, sk) \leftarrow K$ for the operation of executing K and letting (pk, sk) be the pair of keys returned.

•The encryption algorithm E takes the public key *pk* and a plaintext (also called a message) *M* to return a value called the ciphertext. The algorithm may be randomized, but not stateful. We write $C \leftarrow E_{pk}(M)$ or C ← E(pk, M) for the operation of running E on inputs pk, M and letting C be the ciphertext returned.

•The deterministic decryption algorithm D takes the secret key sk and a ciphertext $C \neq \perp$ to return a message *M*. We write $M \leftarrow D_{sk}(C)$ or M ← D (*sk, C).* The message space associated to a public key *pk* is the set Plaintexts (*pk*) of all *M* for which $E_{pk}(M)$ never returns ⊥. We require that the scheme provide correct decryption, which means that for any key-pair *(pk, sk)* that might be output by *K* and any message $M \in \text{Plaintexts}(pk)$, if C was returned by $E_{pk}(M)$ then $D_{sk}(C) = M$.

Let *R* be an entity that wants to be able to receive encrypted communications. The first step is key generation: *R* runs *K* to generate a pair of keys *(pk, sk*) for itself. Note the key generation algorithm is run locally by *R*. Anyone in possession of *R*'s public key pk can then send a message *M* privately to *R*. To do this, they would encrypt *M* via $C \leftarrow E_{pk}(M)$ and send the ciphertext *C* to *R*. The latter will be able to decrypt *C* using sk via $M \leftarrow D_{sk}(C)$.

It is noted that an entity wishing to send data to *R* must be in possession of *R*'s public key *pk,* and must be assured that the public key is authentic, meaning really is the *R*'s public-key, and not someone else's public key. We will look later into mechanisms for assuring this state of knowledge. But the key management processes are not part of the asymmetric encryption scheme itself. In constructing and analyzing the security of asymmetric encryption schemes, we make the assumption that any prospective sender is in possession of an authentic copy of the public key of the receiver.

This assumption is made in what follows. A viable scheme of course requires some security properties. But these are not our concern now. First we want to pin down what constitutes a specification of a scheme, so that we know what are the kinds of objects whose security we want to assess. The key usage is the "mirror-image" of the key usage in a digital signature scheme. In an asymmetric encryption scheme, the holder of the secret key is a receiver, using the secret key to decrypt ciphertexts sent to it by others. In a digital signature scheme, the holder of the secret key is a sender, using the secret key to tag its own messages so that the tags can be verified by others.

The last part of the definition claims that ciphertexts that were correctly generated will decrypt correctly. The encryption algorithm might be randomized, and must for security. But unlike in a symmetric encryption scheme, we will not consider stateful asymmetric encryption algorithms. This is because there is no unique sender to maintain state; many different entities are sending data to the receiver using the same public key. The decryption algorithm is deterministic and stateless.

We do not require that the message or ciphertext be strings. Many asymmetric encryption schemes are algebraic or number-theoretic, and in the natural formulation of these schemes messages might be group elements and ciphertexts might consist of several group elements. However, it is understood that either messages or ciphertexts can be encoded as strings wherever necessary. (The encodings will usually not be made explicit.) In cases where messages are not strings, but group elements for example, using the scheme in practice will usually require encoding of actual messages as group elements. We will discuss this as it arises [24]

2.2.5 Hash Algorithms

Hash algorithms are also known as message digests or one-way transformations. A cryptographic hash function is a mathematical transformation that takes a message of arbitrary length (transformed into a string of bits) and computes from it a fixed-length (short) number.

We'll call the hash of a message m, h(m). It has the following properties:

• For any message *m*, it is relatively easy to compute *h(m).* This just means that in order to be practical it can't take a lot of processing time to compute the hash.

• Given *h(m),* there is no way to find an m that hashes to *h(m)* in a way that is substantially easier than going through all possible values of *m* and computing *h(m)* for each one.

• Even though it's obvious that many different values of *m* will be transformed to the same value *h(m)* (as there are many more possible values of *m*), it is computationally infeasible to find two values that hash to the same thing. [20]

2.3 Notions of Security

Security of an encryption scheme (in both cases of symmetric and asymmetric encryption) is supposed to reflect the inability of an adversary, given ciphertexts (and any public information such as a public key), to get "non-trivial" information about the underlying plaintexts. We allow the adversary (having the goal of figuring out some non-trivial information about plaintexts from ciphertexts) different attack capabilities reflecting different situations. The most basic kind of attack is a *chosen-plaintext attack*, in which the adversary can obtain encryptions of messages of its choice. [24]

2.3.1 Security Against chosen-plaintext attack

Let us fix a specific asymmetric encryption scheme

AE= (K,E,D). We consider an adversary A that is an algorithm (program) that is given as input a public key pk. The intuition behind the notion is as follows. Imagine that the sender has two sequences of messages, $M^1_0, \ldots, M^q_0$ and $M^1_1, \ldots, M^q_1$. It encrypts the messages in one of the sequences to get a sequence of ciphertexts which it transmits. That is, if $b \in \{0,1\}$ denotes the choice of sequence, the sender computes $C^i \leftarrow E_{pk}(M^i_b)$ for $i = 1, \ldots, q$, and then transmits $C^1, \ldots, C^q$ to the receiver. The adversary, being able to eavesdrop, obtains the ciphertexts. Its goal is to figure out which of the two message sequences was encrypted, namely to figure out the value of the bit *b*. The scheme is said to be "secure" if it cannot compute the value of *b* correctly with probability significantly more than *1/2*.

The formalization allows the adversary to specify both message sequences, and furthermore to mount an adaptive attack, meaning to choose M^i_0, M^{i-1}_0 as a function of $C^1, \ldots, C^{i-1}$.

The formalization is in terms of the left-or-right encryption oracle. It depends on the public key and challenge bit *b*. It takes input two messages and returns a ciphertext, as follows:

Oracle $E_{pk}(\mathbf{LR}(M_0, M_1, b))$ // $b \in \{0, 1\}$ and $M_0, M_1 \in \{0, 1\}^*$

If $|M0| \neq |M1|$ *then return* $\perp$

$C \leftarrow E_K(M_b)$

Return *C*

Thus the oracle encrypts one of the messages, the choice of which being made according to the bit *b*. Now we consider two "worlds":

World 0:

The oracle provided to the adversary is $E_{pk}(\mathbf{LR}(., . ,0))$. So, whenever the adversary makes a query (M_0, M_1) to its oracle, the oracle computes $C \leftarrow E_{pk}(M_0)$, and returns C as the answer.

World 1:

The oracle provided to the adversary is $E_{pk}(\mathbf{LR}(., . ,1))$. So, whenever the adversary makes a query (M_0, M_1) to its oracle, the oracle computes $C \leftarrow E_{pk}(M_1)$, and returns *C* as the answer.

We call the first world (or oracle) the "left" world (or oracle), and we call the second world (or oracle) the "right" world (or oracle). The problem for the adversary is, after talking to its oracle for some time, to tell which of the two oracles it was given.

The adversary makes some number of queries to its oracle, and then outputs a bit. This bit has some probability of equaling one. The probability is over the choice of the keys *(pk, sk)* as made by the key-generation algorithm, any random choices made by the oracle, and any other random choices made by the adversary in its computation. We look at this probability in each of the two worlds as the basis for the definition.

Let *AE = (K, E, D)* be an asymmetric encryption scheme, let $b \in \{0,1\}$, and let *A* be an algorithm that has access to an oracle and returns a bit. We consider the following experiment:

Experiment $\boldsymbol{Exp}_{AE}^{ind\text{-}cpa\text{-}b}(A)$

$(pk, sk) \leftarrow K$

$b' \leftarrow A^{E}\ pk$

$(\boldsymbol{LR}(.,.,b))(pk)$

Return b'

The *ind-cpa-advantage* of A is defined as:

$\boldsymbol{Adv}_{AE}^{ind\text{-}cpa}(A) = Pr[\boldsymbol{Exp}_{AE}^{ind\text{-}cpa\text{-}1}(A) = 1] - Pr[Exp_{AE}^{ind\text{-}cpa\text{-}0}(A) = 1.$

As usual, the time-complexity mentioned above is the worst case total execution time of the entire experiment. This means the adversary complexity, defined as the worst case execution time of A plus the size of the code of the adversary A, in some fixed RAM model of computation (worst case means the maximum over A's coins or the answers returned in response to A's oracle queries), plus the time for other operations in the experiment, including the time for key generation and the computation of answers to oracle queries via execution of the encryption algorithm.

Another convention we make is that the length of a query M_0, M_1 to a left-or-right encryption oracle is defined as $|M_0|$. (We can assume without loss of generality that this equals $|M_1|$ since otherwise the oracle returns $\perp$ and so the query would be useless.) The total message length, which is the sum of the lengths of all oracle queries, is another parameter of interest. We say that the total message length is at most μ if it is so in the worst case, meaning across all coin tosses and answers to oracle queries in the experiment.

We consider an encryption scheme to be "secure against chosen-plaintext attack" if a "reasonable" adversary cannot obtain "significant" advantage, where reasonable reflects its resource usage. The technical notion is called indistinguishability under chosen-ciphertext attack, denoted ***IND-CPA***. [24]

2.3.2 Security against chosen-ciphertext attack

Stories introducing chosen-ciphertext attack can be somewhat whimsical. One is about the so-called "lunchtime attack." Entity *R* goes to lunch while leaving his console accessible. For the short period of the lunch break, an adversary gets access to this console; when the lunch break is over, the adversary has to leave before it is discovered at the console by the legitimate user, returning from lunch. The access is such that the adversary cannot actually read the secret decryption key *sk* but does have the capability of executing the algorithm $D_{sk}(.)$ on input any ciphertext of its choice. At that time if the adversary has in his possession some ciphertext it wants to decrypt, it can certainly do so as he cannot be prevented in any manner.

However, it may be able to do even more. For example, perhaps there is some clever sequence of calls to $D_{sk}(.)$ via which the latter can be made to output *sk* itself. (These calls would not be made under normal execution of the algorithm on normal ciphertexts, but the adversary concocts weird ciphertexts that make the decryption routine do strange things.) Having *sk* means the adversary could decrypt traffic at any time in the future, even after the lunch break. Alternatively, the adversary is able to call $D_{sk}(.)$ on some inputs that result in the adversary's gaining some information that would enable it to decrypt some fraction of ciphertexts it might see later, after the lunch break, when it no longer has access to $D_{sk}(.)$. These are the eventualities we want to prevent.

This scenario is artificial enough that were it the only motivation, it would be natural to wonder whether it is really worth the trouble to design schemes to withstand chosen-ciphertext attack. But this is not the main motivation. The real motivation arises from gathering evidence that asymmetric encryption schemes secure against chosen-ciphertext attack are the desired and appropriate tool for use in many higher level protocols, for example protocols for authenticated session key exchange.

There a party decrypts a random challenge message to prove its identity. This leaves it open to a chosen-ciphertext attack on the part of an adversary who sends ciphertexts in the guise of challenges and obtains their decryption. Were this attack to reveal the secret key, the adversary could impersonate the legitimate entity at a later date, since it would now itself possess the ability to decrypt the challenges sent by others.

Based on this and other such applications, we would like to design asymmetric encryption schemes that are secure against very strong kinds of chosen-ciphertext attack. To illustrate let's consider the following

game. An adversary A is given a challenge ciphertext *C* and must output the corresponding plaintext to win the game. The adversary is given the public key *pk* under which *C* was created, and is also given access to the oracle $D_{sk}(.)$ allowing decryption under the secret key *sk* corresponding to *pk*. A trivial way for the adversary to win the game is to invoke its oracle on *C*. This triviality is the one thing disallowed. We allow the adversary to invoke $D_{sk}(.)$ on any input $C' \neq C$. Of course it may invoke the oracle multiple times; all the in puts provided to the oracle must however be different from *C*. If from the information so gathered the adversary can compute $D_{sk}(C)$ then it wins the game.

This is a very strong form of chosen-ciphertext attack: the adversary can invoke the decryption oracle on any point other than the challenge. It is fact that in proving the security of authenticated key exchange protocols that use asymmetric encryption as discussed above, it is exactly security under such an attack that is required of the asymmetric encryption scheme. The other reasons is perhaps more fundamental.

We have seen many times that it is difficult to anticipate the kinds of attacks that can arise. It is better to have an attack model that is clear and well defined even if perhaps stronger than needed, than to not have a clear model or have one that may later be found to be too weak.

We remark that inability to decrypt a challenge ciphertext is not evidence of security of a scheme, since one must also consider loss of partial information. In finalizing a notion of security against chosen-ciphertext attack one must take this into account too. This can be accomplished via left-or-right encryption oracles.

Let AE= (K,E,D) be an asymmetric encryption scheme, let b ∈ {0,1}, and let A be an algorithm that has access to two oracles and returns a bit. We consider the following experiment:

Experiment $\boldsymbol{Exp}_{AE}^{ind\text{-}cca\text{-}b}(A)$

(pk, sk) ← K

$b' \leftarrow A^{Epk(\mathbf{LR}(.,.,b)),\ Dsk(.)}(pk)$

If A queried $D_{sk}(.)$ on a ciphertext previously returned by EK(LR(,, b))

then return 0 else

Return b′

The *ind-cca-advantage* of A is defined as:

$\mathbf{Adv}_{AE}^{ind\text{-}cca}(A) = Pr\,[Exp_{AE}^{ind\text{-}cca\text{-}1}(A) = 1] - Pr[Exp_{AE}^{ind\text{-}cca\text{-}0}(A) = 1].$

The conventions with regard to resource measures are the same as those used in the case of chosen-plaintext attacks.

We consider an encryption scheme to be “secure against chosen-ciphertext attack” if a “reasonable” adversary cannot obtain “significant” advantage, where reasonable reflects its resource usage. The technical notion is called indistinguishability under chosen-ciphertext attack, denoted ***IND-CCA***. [24]

2.4 Semantic Security

In this section we describe an alternative notion of encryption-scheme security, semantic security, under a chosen-plaintext attack. We will abbreviate this notion as SEM-CPA. It captures the idea that a secure encryption scheme should hide all information about an unknown plaintext. Semantic security, which was introduced by Goldwasser and Micali for public-key encryption, transfers the intuition of Shannon’s notion of security to a setting where security is not absolute but dependent on the computational effort made by an adversary. Shannon says that an encryption scheme is secure if *that which can be determined about a plaintext from its ciphertext can be determined in the absence of the ciphertext.* Semantic security asks that *that which can be efficiently computed about some plaintexts from their ciphertexts can be computed, just as easily, in the absence of those ciphertexts.*

Our formalization allows an adversary to choose a message space *M* from which messages may be drawn, and to specify a function *f* on messages. Messages *M* and *M′* are drawn independently and at random from the message space *M*.

We consider two worlds. In the first, the adversary will attempt to compute *f (M)* given an encryption of *M*. In the second, the adversary will attempt to compute *f(M)* given an encryption of *M′* (that is, the

adversary is given no information related to *M*). The scheme is secure if it succeeds about as often in the second game as the first, regardless of the specific (reasonable) f and M that the adversary selects.

To make our definition as general as possible, we will actually let the adversary choose, in sequence, message spaces $M_1, \ldots, M_q$. From each message space M_i draw the message M_i at random, and then let C_i be a random encryption of M_i. Adaptively querying, the adversary obtains the vector of ciphertexts $(C_1, \ldots, C_q)$. Now the adversary tries to find a function *f* such that it can do a good job at predicting $f(M_1, \ldots, M_q)$. Doing a good job means predicting this value significantly better than how well the adversary would predict it had it been given no information about $M_1, \ldots, M_q$: each C_i was not the encryption of M_i but the encryption of a random point M'_I from M_i. The formal definition of Semantic Security now follows.

Let SE = (K,E,D) be a symmetric encryption scheme, and let A be an algorithm that has access to an oracle. We consider the following experiments:

Experiment $\mathbf{Exp}_{SE}^{ss\text{-}cpa\text{-}1}$ (A)	Experiment $\mathbf{Exp}_{SE}^{ss\text{-}cpa\text{-}0}$ (A)
$K \leftarrow K, s \leftarrow \varepsilon$	$K \leftarrow K, s \leftarrow \varepsilon$
for $i \leftarrow 1$ *to* q *do*	*for* $i \leftarrow 1$ *to* q *do*
$(M_i, s) \leftarrow A(s)$	$(M_i, s) \leftarrow A(s)$
$M_i, M'_i \leftarrow M_i$	$M_i, M'_i \leftarrow M_i$
if $\lvert M_i \rvert \neq \lvert M'_i \rvert$ *then* $M_i \leftarrow M'_i \leftarrow \varepsilon$	*if* $\lvert M_i \rvert \neq \lvert M'_i \rvert$ *then* $M_i \leftarrow M'_i \leftarrow \varepsilon$
$Ci \leftarrow E_K(M_i); s \leftarrow (s, C_i)$	$Ci \leftarrow E_K(M_i); s \leftarrow (s, C_i)$
(f, Y) ← A(s)	(f, Y) ← A(s)
return $f(M_1, \ldots, M_q) = Y$	return $f(M_1, \ldots, M_q) = Y$

The SEM-CPA advantage of A is defined as $\boldsymbol{Adv}_{SE}^{sem\text{-}cpa}$ (A) = Pr [$\mathbf{Exp}_{SE}^{ss\text{-}cpa\text{-}1}$ (A) ⇒ 1] – Pr [$\mathbf{Exp}_{SE}^{ss\text{-}cpa\text{-}0$(A) ⇒ 1].

In the definition above, each experiment initializes its oracle by choosing a random key *K*. A total of *q* times, the adversary chooses a message space M_i. The message space is specified by an always-halting probabilistic algorithm, written in some fixed programming language. The code for this algorithm is what the adversary actually outputs. Each time the message space is output, two random samples are drawn from this message space, M_i and M'_i. We expect that M_i and M_I to have the same length, and if they don't we "erase" both strings. The encryption of one of these messages will be returned to the adversary. Which string gets encrypted depends on the experiment:

- M_i for experiment 1 and
- M'_I for experiment 0.

By *f* we denote a deterministic function. It is described by an always-halting program and, as before, it actually the program for *f* that the adversary outputs. By *Y* we denote a string. The string *s* represents saved state that the adversary may wish to retain.

In speaking of the running time of *A*, we include, beyond the actual running time, the maximal time to draw two samples from each message space M that *A* outputs, and we include the maximal time to compute $f(M_1, \ldots, M_q)$ over any vector of strings. In speaking of the length of *A*'s queries we sum, over all the message spaces output by *A*, the maximal length of a string *M* output with nonzero probability by M, and we sum also over the lengths of the encodings of each messages space, function *f*, and string *Y* output by *A*.

We emphasize that the above would seem to be an exceptionally strong notion of security. We have given the adversary the ability to choose the message spaces from which each message will be drawn. We have let the adversary choose the partial information about the messages that it finds convenient to predict. We have let the adversary be fully adaptive. We have built in the ability to perform a chosen-message attack (simply by producing an algorithm M that samples one and only one point). Despite all this, we now cite the theorem claiming that security in the indistinguishability sense implies semantic security (***IND-CPA ⇒ SEM-CPA***).

Let *SE= (K, E, D)* be a symmetric encryption scheme and let A be an adversary (for attacking the SEM-CPA security of SE) that runs in time at most *t* and asks at most *q* queries, these queries totaling at most μ

bits. Then there exists and adversary *B* (for attacking the IND-CPA security of *SE*) that achieves advantage $\boldsymbol{Adv}_{SE}^{ind\text{-}cpa}(B) \geq \boldsymbol{Adv}_{SE}^{sem\text{-}cpa}(A)$ and where *B* runs in time $t + O(\mu)$ and asks at most *q* queries, these queries totaling μ bits.

Proof:

The adversary *B* , which has oracle *g*, is constructed as follows.

Algorithm B^g

$s \leftarrow \varepsilon$
for $i \leftarrow 1$ *to q do*
$(M_i, s) \leftarrow A(s)$
$M_i, M'_i \leftarrow M_i$
if $|M_i| \neq |M'_i|$ then $M_i \leftarrow M'_i \leftarrow \varepsilon$
$C_i \leftarrow g(M'_i, M_i)$, $s \leftarrow (s, C_i)$
$(f, Y) \leftarrow A(s)$
if $f(M1, \ldots, Mq) = Y$ then return 1 else return 0.

Suppose first that *g* is instantiated by a right encryption oracle—an oracle that returns $C \leftarrow E_K(M)$ in response to a query (M', M). Then the algorithm above coincides with experiment $\mathbf{Exp}_{SE}^{ss\text{-}cpa\text{-}1}(A)$. Similarly, if *g* is instantiated by a left encryption oracle—the oracle it returns $C \leftarrow E_K(M')$ in response to a query (M′, M) - then the algorithm above coincides with experiment $\boldsymbol{Exp}_{SE}^{ss\text{-}cpa\text{-}0}(A)$. It follows that $\boldsymbol{Adv}_{SE}^{sem\text{-}cpa}(B) = \boldsymbol{Adv}_{SE}^{ind\text{-}cpa}(A)$. To complete the theorem, note that *B*'s running time is *A*'s running time plus $O(\mu)$ and *B* asks a total of *q* queries, these having total length at most the total length of *A*'s queries, under our convention [21]

3. Related Work

The needs of contemporary life impose cryptography utilization in applications that manage sensitive data. Thus, there is the inclination to embed cryptographic techniques in android applications, despite the fact that the usage of safe cryptography and cryptographic practices has not been yet established. In the specific area of cryptographic techniques use in programming, a limited number of studies and researches have been conducted. Nonetheless, this section provides a synoptic overview of previous work realized in this specific domain, which includes the white papers of a static analysis automated tool, a dynamic analysis automated tool and two automated tools for both static and dynamic analyses.

The first methodical attempt that constitutes a key milestone in the specific domain is Manuel Egele's et al. study [25], the main purpose of which was to test whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security (e.g. IND-CPA security). In order to estimate the legitimacy of cryptographic practices employed in general, the authors developed program analysis techniques to automatically check programs on the Google Play marketplace (CryptoLint project). CryptoLint identifies benign Android applications employing incorrect cryptographic primitives. Its operation is based on static program slicing, aiming to identify flows between IVs, cryptographic keys, and other cryptographic primitives and operations. The identification of cryptographic techniques misuse is realized by analyzing compiled Android applications, having no access to the source code.

The tool was evaluated by testing 145,095 Android applications, downloaded from Google Play marketplace. This examination's result was that 10,327 out of 11,748 applications that use cryptographic APIs (i.e. 88% overall) make at least one mistake. Nevertheless, CryptoLint lacks the capability of analyzing cryptographic primitives' invocation from native code (i.e. code written in other language than Java as C and C++), as its functionality focuses on Dalvik bytecode investigation. CryptoLint also does not include the identification of all types of non-predictable IVs, as the static IV's recognized by the tool refer to a subcategory of non-predictable IVs.

What is more, the tendency to errors (e.g. false alarms) remains a basic drawback of automated tools [26]. Thus, manual static analysis seems to be a more proper approach, guaranteeing more accurate results as well as the ability to cover a greater extent of cryptographic rules. Last but not least, Brumley's paper does not encompass the specific list of android applications checked, neither the tool's code is available so as to be able to repeat the experiments.

Yong Li at al. introduced iCryptoTracer [27], a tool similar to CryptoLint though its function is based on a combination of both static and dynamic analysis techniques. The specific study aims at designing and developing an efficient and effective approach to assess whether the application fulfills basic security goals. ICryptoTracer, in other words, is a cryptography diagnosis system and was designed for diagnosing specifically the implementation code of iOS applications, with the intention to assure that the proper validity of cryptographic usage is in demand.

During static analysis, iCryptoTracer scans and records the cryptographic function APIs locations in the application and later, in the time of dynamic analysis phase, the tool - adopting the message swizzling method- monitors those API calls at runtime. Those related function calls are redirected to the tracing engine of the tool, and the useful information (e.g. API calls, arguments, return values) is recorded to a specific file. Finally, iCryptoTracer synthesizes the information recorded in the crypto-trace file with its diagnosis engine and decides whether a cryptographic misuse exists or not in the application.

The tool's correct operation was confirmed through examining 98 applications downloaded from Apple App Store. Through this examination was determined that the 64 applications of the dataset contain various degrees of security flaws caused by cryptographic misuse. In spite of illustrating the first dynamic analysis tool for cryptographic techniques usage on iOS applications, the paper provides an insufficient set of rules, according to which is realized the classification of applications to Healthy, Weak or Critical.

A quite similar study has been also conducted by Somak Das et al [28], who systematically compared the APIs of cryptographic libraries across different programming languages (C, C++, Java, Python and Go) and evaluated their potential for misuse, as in this paper the possibility to have data security breaches is considered irrespective of the security of cryptography library in use. On the contrary, it depends on the

manner that developer uses the library and consequently, on the properties of each particular library that encourage or discourage cryptographic misuse.

The purpose was to derive recommendations for library designers to follow so as to reduce this misuse. The paper illustrates the comparison of 6 particular cryptographic libraries (OpenSSL in C, Crypto++ and NaCl in C++, PyCrypto in Python, JCA in Java and Go Crypto package in GO) resulting in NaCl being the safest. The authors also developed a linter tool (pycrypto_lint) which applies to any application using PyCrypto library, checking the source code during runtime in order to detect various misuses of the library. The specific study however does not incorporate a specifically defined method according to which each library was examined, and although the source code of the tool is publically available the paper does not include proper sections concerning the description of system design and implementation and the tool's evaluation.

A literature review of Cryptography on Android Message applications has been presented by Nishika and Rahul Kumar Yadav who surveyed and illustrated the most common and widely used SMS Encryption techniques, inferring that there is the need for an efficient encrypting algorithm [29]. The authors of this study distinguished an algorithm based on static Look UP table and Dynamic Key.

Nishika et al. occasioned by the foresaid algorithm -introduced by Manisha Madhwani- proposed an improvement to lookup table and dynamic algorithm. To elaborate, Manisha's algorithm makes use of symmetric key encryption/decryption, built in android Intents and SMS manager to send and receive messages. As far as Lookup Table is concerned, it constitutes an array replacing runtime computation with a simpler array indexing operation. The savings in terms of processing time can be significant, since retrieving a value from memory is often faster than undergoing an 'expensive' computation or input/output operation, which make Manisha's algorithm more efficient than the rest of cryptographic algorithms in use.

Nonetheless, according Nishika et al., Dynamic Lookup table should be used rather than Static Lookup table. The Dynamic Lookup table followed by LZW compression will require less memory as there is no need to store ASCII values corresponding to all characters and due to compression, the size of actual communication text will also be reduced. The values corresponding to SMS will be fetched at runtime. Although Nishika's study -being based on Manisha's algorithm- illustrates an avant-garde idea, it provides to the public a very high-level description of his proposal, without proceeding to any further vital details concerning the implementation.

The most recent work in the specific field of study is that realized by Shuai et al [30]. In the specific study, the authors firstly define specific models of cryptographic misuse, in which are based so as to build a tool of auto detection (CMA). CMA employs both static and dynamic analysis techniques in order to detect cryptographic vulnerabilities and it is tested in 45 Android Applications downloaded from http://as.baidu.com.

CMA's paper includes a quite satisfying number of cryptographic primitives that have to be taken into consideration in such an analysis, which is something that similar papers lack. Nevertheless, the tool created is not designed to locate all the models of cryptographic misuse mentioned in the paper, as the key management category of flaws is omitted. What is more, Shuai et al. does not include in their results all the models of the classes denoted that can be detected by the tool (missing model S5). Additionally, there are models that according to the results are not violated by any of the applications under examination, which makes the proper functionality of the tool for the specific models as well as the necessity of the specific models doubtful. As a result, there are only results for the trivial cryptographic principles misuses.

As for Shuai's tool function, it initially performs static analysis in order to determine if any encryption APIs are employed. If a cryptographic API is detected then the tool creates the Control Flow Graph and the Call Graph of the application and starts the dynamic analysis based on the API's branch located in the previous step. During the execution, runtime logs are produced which are compared to the already defined models of cryptographic misuse and specify if the application has any vulnerabilities. However, it is not clarified if the comparison takes place automatically or manually. Apart from this, according to CMA's description, we can deduce that misses cases where cryptography is employed but is not included in the specific API (when for example the developer has implemented a custom cryptographic algorithm). This fact also indicates the need for including more models for cryptographic misuse in the list. Last but not

least, it has to be remarked the fact that the applications were not downloaded from the official Android Marketplace.

While the majority of the related works are based on automated static or dynamic analysis tools, our approach focuses on the manual static analysis of the Android applications source code in combination with dynamic analysis. A tool cannot guarantee a complete and comprehensive analysis while it is possible to miss certain types of flaws. In addition, although automated tools offer the advantage of being able to examine an entire large code base, it is not always feasible to include a sufficient variety of software flaw categories. Consequently, the studies that have included in their evaluation an automated tool, make use of a short/limited list of cryptographic rules misuses. Apart from this limitation however, tools are also susceptible to errors, as mentioned before. Automated static analysis tools have proven to generate a fair number of false positives while on the contrary, compared to automated tools, static analysis is a manual process and in this case the findings can be verified. [26][31]

Taking into consideration all the above mentioned factors, we are seeking to cover a detailed list of a great variety of cryptographic flaws and misuses, something that developers' community lacks, with a view to helping programmers avoid common –and not only- cryptographic misuses. Due to this particular purpose, and taking into account automated tools' inefficiencies, we have chosen to use manual static analysis in combination with dynamic analysis, so as to have more accurate results and to be able to detect every specific rule of our list in Android applications.

4. Encryption in Java and Android

4.1 Java Cryptography Architecture (JCA)

The Java platform strongly supports security implementation, including language safety, cryptography, public key infrastructure, authentication, access control and secure communication.

The JCA is an essential component of the platform, and contains a "provider" architecture as well as a set of APIs for digital signatures, message digests (hashes), certificates and certificate validation, encryption (symmetric/asymmetric block/stream ciphers), key generation and management, and secure random number generation, to name a few. These APIs allow developers to easily integrate security into their application code. The specific architecture bases its structure on the following principles:

- **Implementation independence**: Applications do not need to implement security algorithms. Rather, they can request security services from the Java platform. Security services are implemented in providers, which are plugged into the Java platform via a standard interface. An application may rely on multiple independent providers for security functionality.
- **Implementation interoperability**: Providers are interoperable across applications. Specifically, an application is not bound to a specific provider, and a provider is not bound to a specific application.
- **Algorithm extensibility**: The Java platform includes a number of built-in providers that implement a basic set of security services that are widely used today. However, some applications may rely on emerging standards not yet implemented, or on proprietary services. The Java platform supports the installation of custom providers that implement such services.

Other cryptographic communication libraries available in the JDK use the JCA provider architecture, but are described elsewhere. The Java Secure Socket Extension (JSSE) provides access to Secure Socket Layer (SSL) and Transport Layer Security (TLS) implementations. The Java Generic Security Services (JGSS) (via Kerberos) APIs, and the Simple Authentication and Security Layer (SASL) can also be used for securely exchanging messages between communicating applications.

The JCA within the JDK includes two software components:

1. the framework that defines and supports cryptographic services for which providers supply implementations. This framework includes packages such as java.security, javax.crypto, javax.crypto.spec, and javax.crypto.interfaces.
2. the actual providers such as Sun, SunRsaSign, SunJCE, which contain the actual cryptographic implementations.

   Whenever a specific JCA provider is mentioned, it will be referred to explicitly by the provider's name. [55]

4.2 Java Cryptography Extension (JCE)

The JavaTM Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects.

JCE was previously an optional package to the JavaTM 2 SDK for the versions 1.2.x and 1.3.x of the Standard Edition. However, JCE has been integrated into the Java since the release of Java 2 SDK, v 1.4.

JCE is based on the same design principles found elsewhere in the Java Cryptography Architecture framework utilized by all the cryptography-related security components of the Java 2 platform. It uses the same "provider" architecture, which uses the notion of a *Cryptographic Service Provider*, or "provider"

for short. This term refers to a package or a set of packages that supply a concrete implementation of a subset of the cryptography aspects of the Java Security API.

JCE extends the list of cryptographic services of which a provider can supply implementations. A provider could, for example, contain an implementation of one or more digital signature algorithms and one or more cipher algorithms.

JCE APIs are implemented by Cryptographic Service Providers. Each of these cryptographic service providers implements the Service Provider Interface which specifies the functionalities which needs to be implemented by the service providers. Programmers can plugin any Service Providers for performing cryptographic functionalities provided by JCE. J2SE comes with a default provider named SunJCE.

Thus, a program wishing to use cryptography functionality may simply request a particular type of object, such as a Cipher object, implementing a particular algorithm, such as DES, and get an implementation from one of the installed providers. If an implementation from a particular provider is desired, the program can request that provider by name, along with the algorithm desired.

Each Java 2 SDK installation, as well as all the subsequent versions of Java, has one or more provider packages installed. Each provider package supplies implementations of cryptographic services defined in one or more security components of the Java 2 SDK (including JCE).

Clients may configure their runtimes with different providers, and specify a preference order for each of them. The preference order is the order in which providers are searched for requested algorithms when no particular provider is requested.

Prior to JDK 1.4, the JCE was an unbundled product, and as such, the JCA and JCE were regularly referred to as separate, distinct components. As JCE is now bundled in the JDK, the distinction is becoming less apparent. Since the JCE uses the same architecture as the JCA, the JCE should be more properly thought of as a part of the JCA. [55]

4.3 Cryptographic Service Providers

java.security.Provider is the base class for all security providers. Each CSP contains an instance of this class which contains the provider's name and lists all of the security services/algorithms it implements. When an instance of a particular algorithm is needed, the JCA framework consults the provider's database, and if a suitable match is found, the instance is created.

Providers contain a package or a set of packages that supply concrete implementations for the advertised cryptographic algorithms. Each JDK installation has one or more providers installed and configured by default. Additional providers may also be added statically or dynamically. Clients may configure their runtime environment to specify the provider preference order. The preference order is the order in which providers are searched for requested services when no specific provider is requested.

To use the JCA, an application simply requests a particular type of object, such as a MessageDigest, and a particular algorithm or service, such as the "MD5" algorithm, and gets an implementation from one of the installed providers. Alternatively, the program can request the objects from a specific provider. Each provider has a name used to refer to it.

```
digest = MessageDigest.getInstance("MD5");
digest = MessageDigest.getInstance("MD5", "ProviderX");
```

The following figure___ illustrates requesting an "MD5" message digest implementation. The specific figure show three different providers that implement various message digest algorithms ("SHA-1", "MD5", "SHA-256", and "SHA-512"). The providers are ordered by preference from left to right. In the first illustration, an application requests an MD5 algorithm implementation without specifying a provider name. The providers are searched in preference order and the implementation from the first provider supplying that particular algorithm, ProviderB, is returned. In the second illustration, the application requests the MD5 algorithm implementation from a specific provider, ProviderC. This time the

implementation from ProviderC is returned, even though a provider with a higher preference order, ProviderB, also supplies an MD5 implementation.

Cryptographic implementations in the JDK are distributed via several different providers (Sun, SunJSSE, SunJCE, and SunRsaSign) primarily for historical reasons, but to a lesser extent by the type of functionality and algorithms they provide. Other Java runtime environments may not necessarily contain these Sun providers, so applications should not request an provider-specific implementation unless it is known that a particular provider will be available.

The JCA offers a set of APIs that allow users to query which providers are installed and what services they support. This architecture also makes it easy for end-users to add additional providers. Many third party provider implementations are already available. [55]

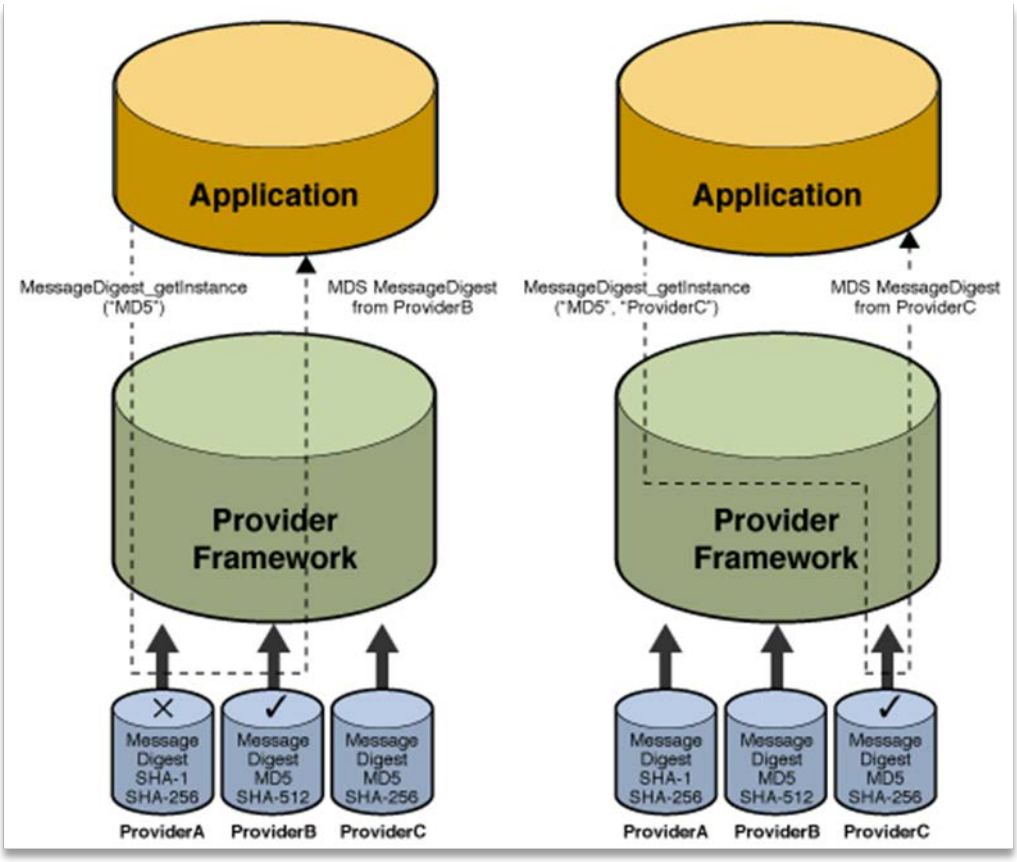

Figure 21: MD5 Message Digest Implementation

4.4 JCA and JCE Design Principles

As already mentioned, the JCA was designed around these principles:

- Implementation independence and interoperability
- Algorithm independence and extensibility

Implementation independence and algorithm independence are complementary; you can use cryptographic services, such as digital signatures and message digests, without worrying about the implementation details or even the algorithms that form the basis for these concepts. While complete algorithm-independence is not possible, the JCA provides standardized, algorithm-specific APIs. When

implementation-independence is not desirable, the JCA lets developers indicate a specific implementation.

Algorithm independence is achieved by defining types of cryptographic "engines" (services), and defining classes that provide the functionality of these cryptographic engines. These classes are called engine classes, and examples are the MessageDigest, Signature, KeyFactory, KeyPairGenerator, and Cipher classes.

Implementation independence is achieved using a "provider"-based architecture. The term Cryptographic Service Provider (CSP) (used interchangeably with "provider" in this document) refers to a package or set of packages that implement one or more cryptographic services, such as digital signature algorithms, message digest algorithms, and key conversion services. A program may simply request a particular type of object (such as a Signature object) implementing a particular service (such as the DSA signature algorithm) and get an implementation from one of the installed providers. If desired, a program may instead request an implementation from a specific provider. Providers may be updated transparently to the application, for example when faster or more secure versions are available.

Implementation interoperability means that various implementations can work with each other, use each other's keys, or verify each other's signatures. This would mean, for example, that for the same algorithms, a key generated by one provider would be usable by another, and a signature generated by one provider would be verifiable by another.

Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can be added easily. [55]

4.5 Java APIs providing security

Java provides many different APIs for implementing encryption in Android:

- javax.crypto
- javax.crypto.interfaces
- javax.crypto.spec
- java.security
- java.security.interfaces
- java.security.spec

The first package provides all the necessary classes and interfaces for cryptographic operations, implementing algorithms for encryption, decryption and key agreement. The javax.crypto.interfaces package provides the specific algorithms that are required to implement key agreement algorithm. The third package is used in order to provide all the necessary classes for keys and other parameters specification.

As for the java.security package, it provides the classes and interfaces for the security framework, such as SecureRandom class, and implements the Extensible cryptographic service provider infrastructure (SPI) for using and defining services such as Certificates, Keys, KeyStores, MessageDigests, and Signatures, while java.security.spec provides the classes and interfaces needed to specify keys and parameters for encryption and signing algorithms. Last but not least, the java.security.interfaces API provides the interfaces needed to generate:

- keys for the RSA asymmetric encryption algorithm using the PKCS#1 standard
- keys for the Digital Signature Algorithm (DSA) and
- keys for a generic Elliptic Curve asymmetric encryption algorithm.

In the following paragraphs we will study in sufficient detail the most significant classes of each package. [55]

4.5.1 The javax.crypto package

This package provides the classes and interfaces for cryptographic applications implementing algorithms for encryption, decryption, or key agreement.

Stream ciphers are supported as well as asymmetric, symmetric and block ciphers. Cipher implementations from different providers can be integrated using the SPI (Service Provider Interface) abstract classes.

Authentication may be based on MAC (Message Authentication Code) such as HMAC (Hash MAC, i.e. with a SHA-1 hash function).

javax.crypto package implements a unique interface:

- **SecretKey** This interface is used for representing cryptographic secret (symmetric) keys.

As for the package's classes, are listed below:

- **Cipher** This class provides access to implementations of cryptographic ciphers for encryption and decryption.
- **CipherInputStream** This class wraps an InputStream and a cipher so that read() methods return data that are read from the underlying InputStream and processed by the cipher.
- **CipherOutputStream** This class wraps an output stream and a cipher so that write methods send the data through the cipher before writing them to the underlying output stream.
- **CipherSpi** This class defines the Service Provider Interface (SPI) for cryptographic ciphers.
- **EncryptedPrivateKeyInfo** This class implements the EncryptedPrivateKeyInfo ASN.1 type as specified in PKCS #8 - Private-Key Information Syntax Standard.
- **ExemptionMechanism** This class implements the functionality of an exemption mechanism such as key recovery, key weakening, or key escrow.
- **ExemptionMechanismSpi** The Service Provider Interface (SPI) definition for the ExemptionMechanism class.
- **KeyAgreement** This class provides the functionality for a key exchange protocol.
- **KeyAgreementSpi** The Service Provider Interface (SPI) definition for the KeyAgreement class.
- **KeyGenerator** This class provides the public API for generating symmetric cryptographic keys.
- **KeyGeneratorSpi** The Service Provider Interface (SPI) definition for the KeyGenerator class.
- **Mac** This class provides the public API for Message Authentication Code (MAC) algorithms.
- **MacSpi** The Service-Provider Interface (SPI) definition for the Mac class.
- **NullCipher** This class provides an identity cipher that does not transform the input data in any way.
- **SealedObject** A SealedObject is a wrapper around a serializable object instance and encrypts it using a cryptographic cipher.

- **SecretKeyFactory** The public API for SecretKeyFactory implementations.
- **SecretKeyFactorySpi** The Service Provider Interface (SPI) definition for the SecretKeyFactory class.

The javax.crypto.interfaces package

This particular package provides the interfaces needed to implement the Diffie-Hellman (DH) key agreement's algorithm as specified by PKCS#3. The parameters for the DH algorithm must be accessed without unduly restriction as for example hardware repository for the parameters material.

The specific interfaces implemented by this package are:

- **DHKey** The interface for a Diffie-Hellman key.
- **DHPrivateKey** The interface for a private key in the Diffie-Hellman key exchange protocol.
- **DHPublicKey** The interface for a public key in the Diffie-Hellman key exchange protocol.
- **PBEKey** The interface to a password-based-encryption key.

4.5.2 The javax.crypto.spec package

This package provides the classes and interfaces needed to specify keys and parameter for encryption. The standards listed below are supported:

- PKCS#3 Diffie-Hellman Key Agreement standard;
- FIPS-46-2 Data Encryption Standard (DES);
- PKCS#5 Password-Based Cryptography standard.

Keys may be specified via algorithm or in a more abstract and general way with ASN.1. Keys and algorithm parameters are specified for the following procedures: *DH, DES, TripleDES, PBE, RC2*, and *RC5*.

The classes of the package are the following:

- **DESedeKeySpec** The key specification for a triple-DES (DES-EDE) key.
- **DESKeySpec** The key specification for a DES key.
- **DHGenParameterSpec** The algorithm parameter specification for generating Diffie-Hellman parameters used in Diffie-Hellman key agreement.
- **DHParameterSpec** The algorithm parameter specification for the Diffie-Hellman algorithm.
- **DHPrivateKeySpec** The key specification for a Diffie-Hellman private key.
- **DHPublicKeySpec** The key specification for a Diffie-Hellman public key.

- **GCMParameterSpec** Provides the parameters for an instance of a Cipher using Galois/Counter Mode (GCM).
- **IvParameterSpec** The algorithm parameter specification for an initialization vector.
- **OAEPParameterSpec** The algorithm parameter specification for the OAEP Padding algorithm.
- **PBEKeySpec** The key specification for a password based encryption key.
- **PBEParameterSpec** The algorithm parameter specification for a password based encryption algorithm.
- **PSource** The source of the label L as specified in PKCS #1.
- **PSource.PSpecified** The explicit specification of the parameter P used in the source algorithm.
- **RC2ParameterSpec** The algorithm parameter specification for the RC2 algorithm.
- **RC5ParameterSpec** The algorithm parameter specification for the RC5 algorithm.
- **SecretKeySpec** A key specification for a SecretKey and also a secret key implementation that is provider-independent.

4.5.3 The java.security package

The package implements the extensible cryptographic service provider infrastructure (SPI) for using and defining services such as Certificates, Keys, KeyStores, MessageDigests, and Signatures.

The most significant interfaces implemented by java.security package are the following:

- **Guard** Guard protects access to other objects.
- **Key** Key is the common interface for all keys.
- **KeyStore.Entry** Entry is the common marker interface for a KeyStore entry.
- **KeyStore.LoadStoreParameter** LoadStoreParameter represents a parameter that specifies how a KeyStore can be loaded and stored.
- **KeyStore.ProtectionParameter** ProtectionParameter is a marker interface for protection parameters.
- **Principal** Principals are objects which have identities.
- **PrivateKey** PrivateKey is the common interface for private keys.
- **PublicKey** PublicKey is the common interface for public keys.

As for java.security classes, the most important of them are cited bellow:

- **AlgorithmParameterGenerator** — AlgorithmParameterGenerator is an engine class which is capable of generating parameters for the algorithm it was initialized with.
- **AlgorithmParameterGeneratorSpi** — AlgorithmParameterGeneratorSpi is the Service Provider Interface (SPI) definition for AlgorithmParameterGenerator.
- **AlgorithmParameters** — AlgorithmParameters is an engine class which provides algorithm parameters.
- **AlgorithmParametersSpi** — AlgorithmParametersSpi is the Service Provider Interface (SPI) definition for AlgorithmParameters.
- **CodeSigner** — CodeSigner represents a signer of code.
- **DigestInputStream** — DigestInputStream is a FilterInputStream which maintains an associated message digest.
- **DigestOutputStream** — DigestOutputStream is a FilterOutputStream which maintains an associated message digest.
- **GuardedObject** — GuardedObject controls access to an object, by checking all requests for the object with a Guard.
- **Identity** — *This class was deprecated in API level 1. Use* Principal, KeyStore *and the* java.security.cert *package instead.*
- **IdentityScope** — *This class was deprecated in API level 1. Use* Principal, KeyStore *and the* java.security.cert *package instead.*
- **KeyFactory** — KeyFactory is an engine class that can be used to translate between public and private key objects and convert keys between their external representation, that can be easily transported and their internal representation.
- **KeyFactorySpi** — KeyFactorySpi is the Service Provider Interface (SPI) definition for KeyFactory.
- **KeyPair** — KeyPair is a container for a public key and a private key.
- **KeyPairGenerator** — KeyPairGenerator is an engine class which is capable of generating a private key and its related public key utilizing the algorithm it was initialized with.
- **KeyPairGeneratorSpi** — KeyPairGeneratorSpi is the Service Provider Interface (SPI) definition for KeyPairGenerator.
- **KeyRep** — KeyRep is a standardized representation for serialized Key objects.
- **KeyStore** — KeyStore is responsible for maintaining cryptographic keys and their owners.
- **KeyStore.Builder** — Builder is used to construct new instances of KeyStore.
- **KeyStore.CallbackHandlerProtection** — CallbackHandlerProtection is a ProtectionParameter that encapsulates a CallbackHandler.

- **KeyStore.PasswordProtection** PasswordProtection is a ProtectionParameter that protects a KeyStore using a password.
- **KeyStore.PrivateKeyEntry** PrivateKeyEntry represents a KeyStore entry that holds a private key.
- **KeyStore.SecretKeyEntry** SecretKeyEntry represents a KeyStore entry that holds a secret key.
- **KeyStore.TrustedCertificateEntry** TrustedCertificateEntry represents a KeyStore entry that holds a trusted certificate.
- **KeyStoreSpi** KeyStoreSpi is the Service Provider Interface (SPI) definition for KeyStore.
- **MessageDigest** Uses a one-way hash function to turn an arbitrary number of bytes into a fixed-length byte sequence.
- **MessageDigestSpi** MessageDigestSpi is the Service Provider Interface (SPI) definition for MessageDigest.
- **PolicySpi** Represents the Service Provider Interface (SPI) for java.security.Policy class.
- **ProtectionDomain** Legacy security code; do not use.
- **Provider** Provider is the abstract superclass for all security providers in the Java security infrastructure.
- **Provider.Service** Service represents a service in the Java Security infrastructure.
- **SecureClassLoader** SecureClassLoader represents a ClassLoader which associates the classes it loads with a code source and provide mechanisms to allow the relevant permissions to be retrieved.
- **SecureRandom** This class generates cryptographically secure pseudo-random numbers.
- **SecureRandomSpi** SecureRandomSpi is the Service Provider Interface (SPI) definition for SecureRandom.
- **Security** Security is the central class in the Java Security API.
- **Signature** Signature is an engine class which is capable of creating and verifying digital signatures, using different algorithms that have been registered with the Security class.
- **SignatureSpi** SignatureSpi is the Service Provider Interface (SPI) definition for Signature.
- **SignedObject** A SignedObject instance acts as a container for another object.
- **Timestamp** Timestamp represents a signed time stamp.

4.5.4 The java.security.interfaces package

As mentioned before, this package provides the interfaces needed to generate: keys for the RSA asymmetric encryption algorithm using the PKCS#1 standard, keys for the Digital Signature Algorithm (DSA) and keys for a generic Elliptic Curve asymmetric encryption algorithm.

The aforementioned interfaces are cited bellow:

• **DSAKey**	MessageDigestSpi is the Service Provider Interface (SPI) definition for MessageDigest.
• **DSAKeyPairGenerator**	Represents the Service Provider Interface (SPI) for java.security.Policy class.
• **DSAParams**	Legacy security code; do not use.
• **DSAPrivateKey**	Provider is the abstract superclass for all security providers in the Java security infrastructure.
• **DSAPublicKey**	Service represents a service in the Java Security infrastructure.
• **ECKey**	SecureClassLoader represents a ClassLoader which associates the classes it loads with a code source and provide mechanisms to allow the relevant permissions to be retrieved.
• **ECPrivateKey**	This class generates cryptographically secure pseudo-random numbers.
• **ECPublicKey**	SecureRandomSpi is the Service Provider Interface (SPI) definition for SecureRandom.
• **RSAKey**	Security is the central class in the Java Security API.
• **RSAMultiPrimePrivateCrtKey**	Signature is an engine class which is capable of creating and verifying digital signatures, using different algorithms that have been registered with the Security class.
• **RSAPrivateCrtKey**	SignatureSpi is the Service Provider Interface (SPI) definition for Signature.
• **RSAPrivateKey**	A SignedObject instance acts as a container for another object.
• **RSAPublicKey**	Timestamp represents a signed time stamp.

4.5.5 The java.security.spec package

This package provides the classes and interfaces needed to specify keys and parameters for encryption and signing algorithms. The following standards are supported:

1. PKCS#1 RSA encryption standard;
2. FIPS-186 DSA (signature) standard;
3. PKCS#8 private key information standard. Keys may be specified via algorithm or in a more abstract and general way with ASN.1.

The parameters for the Elliptic Curve (EC) encryption algorithm are only specified as input parameters to the relevant EC-generator.

Java.security.spec interfaces are the following:

• **AlgorithmParameterSpec**	SignatureSpi is the Service Provider Interface (SPI) definition for Signature.
• **ECField**	A SignedObject instance acts as a container for another object.

- **KeySpec** Timestamp represents a signed time stamp.

The particular package also include the classes listed below:

- **DSAParameterSpec** The parameter specification used with the Digital Signature Algorithm (DSA).
- **DSAPrivateKeySpec** The parameters specifying a DSA private key.
- **DSAPublicKeySpec** The parameters specifying a DSA public key.
- **ECFieldF2m** The parameters specifying a characteristic 2 finite field of an elliptic curve.
- **ECFieldFp** The parameters specifying a prime finite field of an elliptic curve.
- **ECGenParameterSpec** The parameter specification used to generate elliptic curve domain parameters.
- **ECParameterSpec** The parameter specification used with Elliptic Curve Cryptography (ECC).
- **ECPoint** A Point on an Elliptic Curve in barycentric (or affine) coordinates.
- **ECPrivateKeySpec** The parameters specifying an Elliptic Curve (EC) private key.
- **ECPublicKeySpec** The parameters specifying an Elliptic Curve (EC) public key.
- **EllipticCurve** An Elliptic Curve with its necessary values.
- **EncodedKeySpec** The abstract key specification for a public or a private key in encoded format.
- **MGF1ParameterSpec** The parameter specification for the Mask Generation Function (MGF1) in the RSA-PSS Signature and OAEP Padding scheme.

4.6 The OpenSSL API

Apart from the already discussed APIs, developers have also the possibility to utilize the OpenSSL API so as to encrypt data. Encrypting data in OpenSSL on Android requires writing native C code that can be accessed in Java through JNI calls. Implementing encryption with OpenSSL API constitutes a more composite procedure. However, OpenSSL offers a much better performance. [56]

4.6.1 Encryption and Decryption Procedures Examples

In this current paragraph we will study the specific procedures of encryption and decryption that should be used in android applications source code, and we will analyze particular classes from the aforementioned APIs.

Regarding encryption with Java Core Classes, an example encryption procedure is cited below:

1. **Encryption Key generation**

Key generation functionality in Java is provided by **KeyGenerator** class of Crypto API. In the following figure is cited the specific code required to be used in order to produce a key with AES cryptographic algorithm.

```
mKey = null;
try {
    kgen = KeyGenerator.getInstance("AES");
    mKey = kgen.generateKey();
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
```

Figure 22: Key Generation with AES

Another way to produce an encryption key in a more secure way is through the combination of **KeyGenerator** and **SecureRandom** classes' utilization. The following figure depicts a function (generateKey) which initializes KeyGenerator with a source of randomness.

```
public static SecretKey generateKey() throws NoSuchAlgorithmException {
    // Generate a 256-bit key
    final int outputKeyLength = 256;

    SecureRandom secureRandom = new SecureRandom();
    // Do *not* seed secureRandom! Automatically seeded from system entropy.
    KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
    keyGenerator.init(outputKeyLength, secureRandom);
    SecretKey key = keyGenerator.generateKey();
    return key;
}
```

Figure 23: Key Generation with AES and SecureRandom

Another approach which is considered even more secure for key generation, is to use a technique called *Password Based Encryption*. To be more specific, in the case that an application needs additional encryption, a recommended approach is to require a passphase or PIN to access the application. This passphrase could initialize a Password Based Key Derivation Function (PBKDF), for example in PBKDF2, in order to generate the encryption key. PBKDF2 is a commonly used algorithm for deriving key material from a passphrase, using a technique known as "key stretching". Android provides an implementation of this algorithm inside SecretKeyFactory as PBKDF2WithHmacSHA1 [57]:

```
public static SecretKey generateKey(char[] passphraseOrPin, byte[] salt) throws NoSuchAlgorithmException, InvalidKeySpecException {
    // Number of PBKDF2 hardening rounds to use. Larger values increase
    // computation time. You should select a value that causes computation
    // to take >100ms.
    final int iterations = 1000;

    // Generate a 256-bit key
    final int outputKeyLength = 256;

    SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    KeySpec keySpec = new PBEKeySpec(passphraseOrPin, salt, iterations, outputKeyLength);
    SecretKey secretKey = secretKeyFactory.generateSecret(keySpec);
    return secretKey;
}
```

Figure 14: Password Based Encryption

2. **Initialization Vector (IV) Creation**
   In particular cases the cryptographic primitives employed for encryption require the IV utilization so as to be non-repeating, and the required randomness is derived internally.
   An example of a proper initialization vector generation is following.

```
cipher.init(Cipher.ENCRYPT_MODE, secret);
IvParameterSpec spec = cipher.getParameters().getParameterSpec(IvParameterSpec.class);
byte[] iv = spec.getIV();
```

Figure 25: Initialization Vector Generation

3. **Encryption procedure**
   The key produced in the first step can now be used in the encryption procedure. An example of a basic encryption procedure for a string is cited in the following figure.

```
public static String encrypt(String Data) throws Exception {
        Key key = generateKey();
        Cipher c = Cipher.getInstance(ALGO);
        c.init(Cipher.ENCRYPT_MODE, key);
        byte[] encVal = c.doFinal(Data.getBytes());
        String encryptedValue = new BASE64Encoder().encode(encVal);
        return encryptedValue;
    }
```

Figure 26: Encryption Procedure

4. **Decryption procedure**
   The equivalent decryption procedure follows:

```
public static String decrypt(String encryptedData) throws Exception {
        Key key = generateKey();
        Cipher c = Cipher.getInstance(ALGO);
        c.init(Cipher.DECRYPT_MODE, key);
        byte[] decordedValue = new BASE64Decoder().decodeBuffer(encryptedData);
        byte[] decValue = c.doFinal(decordedValue);
        String decryptedValue = new String(decValue);
        return decryptedValue;
    }
```

Figure 27: Decryption Procedure

In both second and thirds steps we can notice that in order to call an encryption algorithm the **Cipher** class invoking is required. Effectively, Cipher class takes as input a plaintext and produces a ciphertext and vice versa, taking into considerations the encryption key and other cryptographic parameters. The Cipher class's visualization is depicted in figure 28.

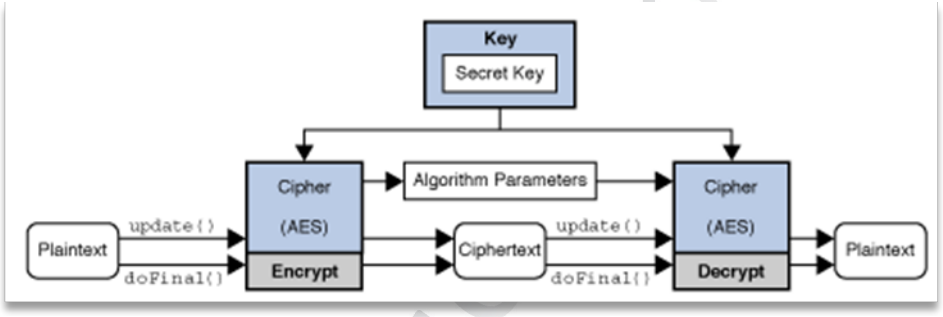

Figure 28: Cipher Class functionality

As far as encryption with OpenSSL API is concerned, below is cited a file encryption example in C language.

```
if (!(EVP_EncryptInit_ex(e_ctx, EVP_aes_256_cbc(), NULL, cKeyBuffer, iv ))) {
    ret = -1;
    printf( "ERROR: EVP_ENCRYPTINIT_EXn");
}

// go through file, and encrypt
if ( orig_file != NULL ) {
    origData = new unsigned char[aes_blocksize];
        encData = new unsigned char[aes_blocksize+EVP_CIPHER_CTX_block_size(e_ctx)]; // potential for encryption to be 16 bytes longer than original
    printf( "Encoding file: %sn", filename);
    bytesread = fread(origData, 1, aes_blocksize, orig_file);

    // read bytes from file, then send to cipher
    while ( bytesread ) {
        if (!(EVP_EncryptUpdate(e_ctx, encData, &len, origData, bytesread))) {
            ret = -1;
            printf( "ERROR: EVP_ENCRYPTUPDATEn");
        }
        encData_len = len;

        fwrite(encData, 1, encData_len, enc_file );
        // read more bytes
        bytesread = fread(origData, 1, aes_blocksize, orig_file);
    }

    // last step encryption
    if (!(EVP_EncryptFinal_ex(e_ctx, encData, &len))) {
        ret = -1;
        printf( "ERROR: EVP_ENCRYPTFINAL_EXn");
    }

    encData_len = len;

    fwrite(encData, 1, encData_len, enc_file );

    // free cipher

    EVP_CIPHER_CTX_free(e_ctx);
```

Figure 29: Encryption using OpenSSL API

5. An Overview of Cryptographic Concepts & Cryptographic Rules for Android Programming

It is fact that the goal of encryption employment is to provide privacy. Nonetheless, applications which employ cryptography in the interest of guaranteeing privacy to their users, can be attacked in many different ways. The most usual way -plus relevant with cryptographic schemes employment per se- is breaking encryption schemes embodied in the application.

This particular class of attacks consists of three basic subcategories: the ciphertext only, the known plaintext and the chosen plaintext attacks. In a ciphertext only attack, the adversary has access to a specific ciphertext which he tries to decrypt searching in the set of all the possible keys, while in a known plaintext attack the attacker has in his possession a pair of plaintext and ciphertext. In a chosen plaintext attack, the adversary can access any possible plaintext with its corresponding ciphertext. [21]

A cryptosystem should resist all the above mentioned sorts of attacks. In our work we will encounter specifically the term ciphertext indistinguishability. This property, also known as Indistinguishability under Chosen Plaintext Attack (IND-CPA), ensures that a potential adversary will not be able to distinguish pairs of ciphertexts based on the message they encrypt.

A secure cryptosystem constitutes any entity employing cryptography, in hardware or software level, which averts the threat of an adversary to discern even a single bit of information describing the plaintext given the ciphertext in polynomial time. Taking this fact into consideration, we should only consider an encryption scheme to be secure if and only if it is IND-CPA secure. Moreover, an encryption scheme must be either probabilistic or stateful to be IND-CPA secure [21]. Otherwise, the adversary will be able to discern if the same message was sent twice. It is noted that a stateful encryption scheme that the keys are updated in each encryption, while a probabilistic encryption scheme is the one that uses randomness in the encryption algorithm, satisfies collision resistance and hides all the information related to its input [32].

As for the existent types of encryption, Password Based Encryption (PBE) is highly widespread in Android Applications. Password Based Encryption is a cryptographic technique with the use of which a secret key is generated based on a user-generated passphrase. This particular technique is proposed to be used with a high entropy password, as PBE is usually used in applications where the adversary is able to apply brute force attack to retrieve the password without being detected [33].

5.1 Cryptographic Rules

The classification of the examined applications to secure and non-secure is based on a list of particular cryptography principles. The techniques responsible for data breaches and security vulnerabilities in applications are cited below and must be strictly avoid by developers.

1. The use of weak cryptographic algorithms or hash functions: Programmers should not use algorithms that are official known to be attacked and broken or weak. For example, MD4, MD5, SHA1, DES and RC4 are considered to be obsolete [34].
2. The use of custom cryptographic algorithms, invented by programmers individually: The security offered by not official algorithms is questionable and their employment is very possible to be insecure [35].
3. Re-implementing the known well-designed algorithms (e.g. AES): Similarly, re-implementations of the known algorithms are also possible to be incorrect and consequently insecure. Thus, the developers should not use other implementations of well-designed algorithms than those introduced by industry standard encryption [35].
4. ECB mode: It does not constitute a secure cryptographic mode, as it cannot be IND-CPA secure [25][28].
5. The use of block ciphers AES, DES and DESede (triple DES) with Java's default cryptographic mode: This particular practice is not considered secure as in Java, at any case, when only the

cipher algorithm is invoked (without a specific mode defined) the default cryptographic mode used in specific providers (SunJCE and SunPKCS11) is the ECB [36][37].

6. The use of RSA with ECB mode: Due to the fact that RSA is a multiplicative homomorphic cipher, an attacker can alter a ciphertext, the plaintext of which is known to be valid, by multiplying it with another ciphertext. For this reason, RSA is sometimes used with padding by developers. What is more, RSA is also sometimes used in modes of operations of symmetric cryptography (block ciphers) –such as CBC- so as to be more difficult for the adversaries to extract parts of the ciphertext and additionally, in order to constrain the ability to create a ciphertext only to the key owner. Nevertheless, ECB is not a proper solution for succeeding in implementing the aforementioned property, as Java does not provide its implementation and effectively, when ECB is called with RSA is discounted [38]. The developers have to implement the ECB mode for RSA, which also does not constitute a safe solution.
7. The combination of CBC encryption mode with a non-random IV: It does not constitute a safe cryptographic scheme. The term "random IV" refers to an initialization vector that is neither static nor predictable (for example an IV consisting of 0's or sequential numbers) [25][27].
8. The combination of CTR encryption mode with a static counter value: It does not constitute a safe cryptographic scheme as it is not IND-CPA secure [28].
9. The use of weak keys: Yet another possible vulnerability of a cryptographic algorithm –apart from improper cryptographic logic and weakness- is weak keys employment. According to the contemporary cryptographic standards [39], a key is recognized as weak when its length is less than 128 bits. The usage of a suchlike cryptographic key weakens the resulting encryption and must be strictly avoided. For example, DES is known to have a set of weak keys, as it uses a 56-bits key, which does not provide sufficient security [34].
10. The use of hard-coded encryption keys: The effectiveness of the specific practice in an application can result even in the disclosure of the key to the adversary and guarantees practically no security [34], as the secrecy of encryption keys is an important factor having impact on encryption schemes security. The encryption keys must be dynamically generated and developers should strictly avoid exposing them in application's code [25][27].
11. The use of static/constant encryption keys: The randomness of the encryption keys is the major factor contributing to encryption schemes security. Cryptographic keys must not be constant [25][27]. It is noted that an encryption key it is possible to be static without being hard-coded, for example when a byte array is initialized and remains the same for the whole process.
12. The use of hard-coded IVs: As a consequence of the tenth rule, the IVs must not be hard-coded. On the contrary, the developers have to generate them dynamically for two specific reasons: preventing adversaries from obtaining the specific primitive's value and generating different values for the IV in each different cryptographic stage [25].
13. Constant IV: A constant IV or an IV reuse renders generally many cryptographic schemes IND-CPA insecure, as the IV constitutes the only primitive introducing randomness in a cryptographic procedure. Namely, using a constant or a static IV frequently results in producing the same ciphertext [28]. Similarly to rule 11, an IV can be constant without being hard-ceded, if for example is randomly generated but used more than one times.
14. Deriving IV from keys or messages: This practice's usage was observed in specific applications of our examination set and is considered to be insecure due to the fact that makes the IV non-random and predictable [25][27].
15. Generating IV from cipher's blocksize, based on byte array creation: Many developers generate manually the IVs by initializing a vector having the size of cipher's blocksize with the default values of the creation of a byte array (bytearray = new byte[]), in combination with nextbytes() method of Random class. There are also cases where not even Random class is utilized. It has to be noted that Random class use is not a proper practice, though it is observed in many applications of our examination set. What is more, it is a given that deriving the IV without introducing any randomness, using Java default values to a byte array, does not also constitute a best practice due to the fact that this practice makes the IV non-random and predictable [25][27].
16. The use of PBE with constant salts: This rule constitutes a best practice for PBE usage, proposed in order to avoid brute force attacks [25].

17. The use of PBE with fewer than 1.000 iterations: This rule also constitutes a best practice for PBE usage, proposed also in order to avoid brute force attacks [25].

18. The use of hard-coded passwords for PBE: Although PBE is usually used based on a password given by the user as an input to the Android Application, there are cases where the developers use a specific value defined statically by them. In this way, developers make the application use the same password for each execution, while the password value can easily be accessed by the adversary.

19. The use of non-Cryptographically Secure PseudoRandom Number Generators (PRNGs): According to official "Android Developers" security guidelines, a cryptographic key that is not generated with a secure random number generator weakens the strength of the cryptographic algorithm. Thus, it is proposed that the developers use proper algorithms for key generation, which introduce sufficient entropy, rendering the guess of future bits a hard problem. Java provides for Android Development the SecureRandom class which implement a PRNG for keys production [34][35]. It is noted that Random class of Java does not introduce adequate randomness for key generation and as a matter of fact, PRNGs that use system entropy in order to seed data usually produce better results, making the initial conditions of the PRNG much more difficult for the adversary to guess [40]. The factor of randomness, however, should be also introduced in any kind of password, salt and seed.

20. The use of predictable PRNG seeds: Apart from the selection of a proper algorithm, the seed of the PRNG also constitutes an important factor in constructing a secure cryptographic scheme, as the use of a PRNG for key generation is not sufficient enough to guarantee a totally secure cryptosystem. Developers should use no predictable seeds with PRNGs so as to generate a high entropy key and so as not to weaken PRNG's strength [34]. Random Class of Java could be used for example in order to achieve the production of a non-predictable seed for the cryptographically secure PRNG. It is also essential to note that the setSeed() method of the SecureRandom class produces a predictable seed and must not be used in the key generation process [41].

21. The reuse of PRNG seed-Use of static PRNG seed: In the same context, the PRNG seed must not be reused as it is a best practice to use independent random numbers in all the cryptographic stages of a cryptographic procedure. Moreover, specifically for the SecureRandom class it is known that a static seed will produce the same PRNG output [25][34].

22. The omission or the improper use of dofinal() function: When Cipher java library is used for cryptography, the proper call of dofinal() function should not be omitted for both the encryption and the decryption phase. The specific function processes the last block in the buffer given as argument (i.e. the ciphertext or the plaintext). The internal mechanism of the algorithm implementation, based in its encryption mode (ECB, CBC, or other) keeps an internal buffer which must be also discarded [42].

23. The utilization of libraries that are known to contain unsafe cryptographic practices: According to relevant studies, there are numerous libraries employing unsafe cryptography affecting also the security of the android application that embeds the specific library [25][28].

24. Finally, regarding the padding techniques, as CBC combined with PKCS5Padding is vulnerable to padding oracle attacks, PKCS7 is considered to be the best option for the specific mode [43][44][45][46][47][48][49].

6. Methodology and Experiments

Our approach is organized in four main phases:

- Applications Collection
- Applications Utilization
- Static Analysis
- Dynamic Analysis.

The first phase describes the procedure of specifying the specific Android applications that will be audited, while the second includes applications' testing through their graphical user interface (GUI). The core of our study however, is detailed in the phases three and four where static and dynamic analyses are conducted with a view to ascertaining possible cryptographic misuses.

6.1 Applications Collection

This initial phase of our study comprises the selection of the specific amount and the particular type of the applications that will be under examination. We chose to analyze 49 Android applications of four different categories related to data encryption:

1. *Secure Messaging:* This category includes applications that exchange encrypted data either via SMS or through Bluetooth and Internet services (chat, social media and email).
2. *Document Encryption:* Document Encryption describes the applications that are involved with any kind of document encryption: files encryption, directories encryption, multimedia content encryption, even notes encryption.
3. *Sensitive Data Exchange & Storage:* Applications that appertain to this particular category are those handling any type of sensitive data (passwords, credit card numbers, pins etc.).
4. *Multipurpose Encryption Utility:* This particular class of applications contains applications of multiple purposes, offering more than one operations, such as generating passwords, document encryption, text encryption, sensitive data storage, password vaults etc.

The applications were downloaded from official Google Play Marketplace between June and November 2014. This particular aggregation of applications was considered to be a representative sample of developers' predilection for certain cryptographic primitives and strategies.

6.2 Applications Test Utilization

Prior to the static and the dynamic analysis and after the collection of applications' .apk files, we installed each and every one of them in at least 2 different android devices. The purpose was to run the applications and test them through their graphical environment so as to recognize any parameters used that are possibly involved in the cryptographic procedures employed. Moreover, in the particular case of the applications that appertain to the Secure Messaging category, we are able to form an opinion regarding the general legitimacy of cryptographic practices employed, as the cipher is directly available via graphical user interface.

The particular parameter checked to the total of the applications is the *utilization of a password.* Applications encompassing encryption usually use a password consisting of alpha, numeric or alphanumeric characters. This parameter is introduced by the user and commonly, takes part in the process of user's plaintext encryption. There are many cases however, where the password is used only as a pin.

The parameter that we checked particularly in Secure Messaging applications was the *production of the same ciphertext* (output) when the same plaintext is given as an input. When this particular finding is detected, we can deduce that the cryptographic scheme used is not IND-CPA secure, as if for a specific plaintext the same ciphertext is computed, the requirement of ciphertext indistinguishability is not met. Namely, the output of the same ciphertext implies the usage of wrong cryptographic primitives, for example the use of the same IV for each encryption. The same stands also in the case of password usage (i.e. if for the same combination of plaintext and password word the same ciphertext is produced).

Finally, it is necessary to be remarked that specific parameter was inspected under three different scenarios, described in the following section.

6.2.1 Experimental Scenarios

In order to ascertain ciphertext's indistinguishability through each application's graphical environment we considered three different cases that would possibly have impact in ciphertext's diversification. Next, are cited the scenarios of the foresaid cases.

1st Scenario

Our first scenario refers to the input of the same plaintext twice in the application under examination, without disrupting its operation.

2nd Scenario

This scenario includes application's execution, entering the same plaintext once before and once after application's and device's reboot.

3rd Scenario

Our last scenario requires the execution of the application in two different android devices, inserting in both cases the same plaintext.

6.3 Applications Analysis

Our specific approach regarding applications analysis was to employ the combination of both techniques of static and dynamic analyses, so as to succeed in producing more accurate results.

Generally, the term *Static Analysis* refers to the process of detecting software errors and defects or security flaws by examining the source code of a program without executing it, and can also be utilized for ensuring conformance with specific programming requirements. Static Analysis is considered as a part of Code Review process and provides a better perception of code structure [26][50][51]. Developers frequently perform static analysis combining automated tools and visual inspection of the source code.

On the other hand, *Dynamic Analysis* refers to the testing and evaluation of a program based on its execution and it is usually performed with a view to detecting subtle defects or vulnerabilities manifested during runtime, the cause of which is too perplex to be detected via static analysis [50][52]. Developers through a dynamic test are capable of monitoring system memory, functional behavior, response time, and overall performance of the system [53]. Therefore, there are cases where a single component from the abovementioned list is selected to be examined (for example the system memory) in order to seek only for specific types of errors.

On a comparison between the advantages and weaknesses of the two methods, it is a given that static analysis has many advantages to offer as it is the most thorough technique. The developers using the specific technique of analysis are capable of identifying the exact location of weaknesses in the code [52] as well as of examining all possible execution paths and variable values and not just those invoked during execution [54]. Last but not least, Static Analysis reveals errors in the initial stages of the development life cycle, reducing the cost to fix [52] and mainly before they manifest themselves and trigger any incident [50].

On the contrary, Dynamic Analysis is more flexible regarding the possibility to test the application only apropos specified error categories, for instance security flaws. What is more, via Dynamic Analysis it is technically feasible to test applications even if there is not access to their source code. Finally, Dynamic Analysis can be utilized as a validation of Static Analysis results [52].

Nevertheless, the two methods of analysis have many disadvantages both due to their nature per se but also due to the fact that the use of automated tools for their employment is widespread. It is fact that in cases where automated tools are utilized, the significant number of false positives and false negatives constitute the main drawback in both types of analyses as the tools' efficiency is highly dependent on the rules defined for scanning the software [52].

This specific fact remarks the necessity for the human factor involvement for understanding whether the tool alerted a real error or not [51].

Additionally, Static Analysis cannot provide satisfactory results regarding memory leaks and concurrency errors. In order to detect this type of faults is necessary to execute the software [51]. Lastly, when Static Analysis is performed by a tool, there is a limitation regarding the programming languages that can be supported.

Consequently, we can deduce that the two approaches are complementary as no single approach can find every possible types of errors [52].

6.3.1 Static Analysis

With the intention of further inspecting, in full detail, the cryptographic algorithms, primitives and strategy in use, we proceeded in the source code auditing during the third stage of our study.

Regarding the specific procedure for our static code analysis and review approach, it begins with the reverse engineering of .apk files. The sense of reverse engineering of an android application, within the context of our work, involves the following steps:

- 1st step - Obtaining target application's .apk file: This particular procedure is necessary to take place in a rooted device. What is more, we downloaded and installed Root File Explorer, so as to be able to explore our device's files and copy the target application's .apk from /data/app/ file to the SD Card. In this manner, we extracted all the .apk files from the Smartphone.
- 2nd step – Extracting the source code from the .apk file: At this point, we used dex2jar tool as well as JD-GUI tool of Java Decompiler project. Specifically, dex2jar tool is used in order to convert .dex files of android to .class files in Java, while JD-GUI is a standalone graphical utility that displays Java source codes of .class files. The steps sequence followed in this stage for each application was the following:
    - Adding the extension ".zip" to the .apk file and extracting the zip file produced.
    - Extracting the contents of zipped dex2jar inside the extracted folder produced in the previous step.
    - Generating classes.dex.dex2jar file through the command line in the same folder.
    - Opening classes.dex.dex2jar file through Java Decompiler. Finally, we have gained access to the source code.
- 3rd step – Source Code Analysis and Reviewing: The current step substantiates the core of our study. Our purpose is to extensively examine all the parameters involved in the encryption-decryption procedures in order to determine if they are properly defined and used. The classification of the examined applications to secure and non-secure is based on the list of particular cryptography principles discussed in previous chapter, which are used to conduct visual inspection in applications source code.

The specific procedure followed to complete static analysis is presented in the following figure.

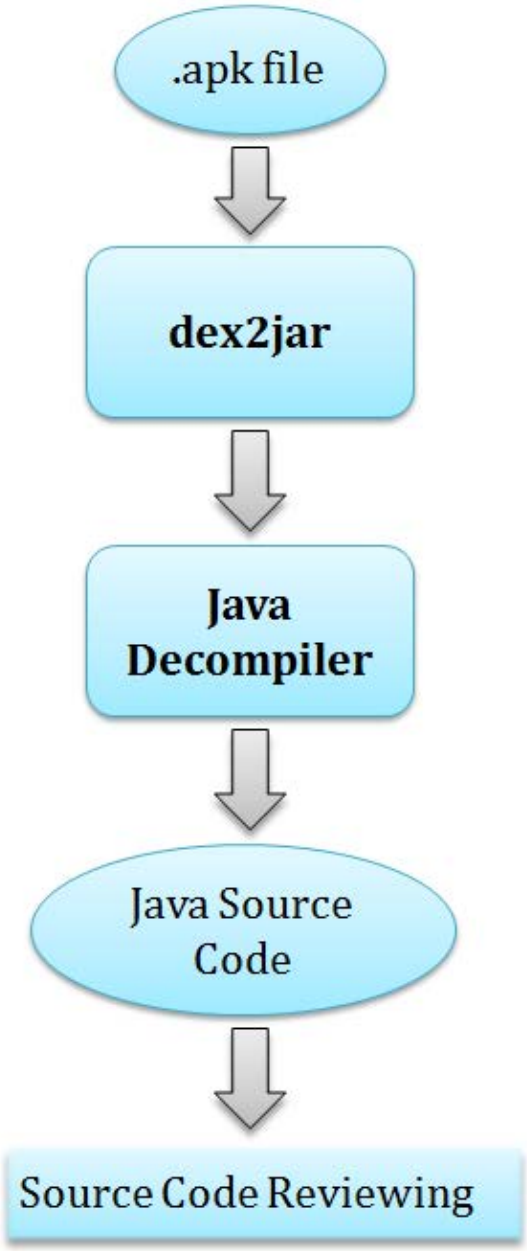

Figure 30: Static Analysis Procedure

6.3.2 Dynamic Analysis

The current phase concerns our approach of dynamic analysis of the total of the applications. This additional examination is conducted in order to ensure the security of the cryptographic primitives in use that are not evident in the source code. Specifically, owing to the fact that many Android applications make use of native code -which is not available after .apk' decompilation process that we followed- static analysis cannot always cover the whole functionality of the application.

What is more, there is always the possibility -due to bad programming techniques employed by the developer- to include to the code functions that are not finally called during application's execution. With

a view to including to our research these cases too, as well as the cases of the applications that do not produce an apparent cipher -such as document encryption applications- so as to examine ciphertext indistinguishability and, seeking to have more accurate results, we combined static analysis with dynamic analysis techniques.

As far as the specific dynamic analysis procedure is concerned, we chose to install and run the android applications under examination to emulators and monitor their functionality through Dalvik Debug Monitor Server (DDMS) of Android SDK. Particularly, via Track Memory Allocation utility we were able to detect the exact cryptographic algorithms and functions executed.

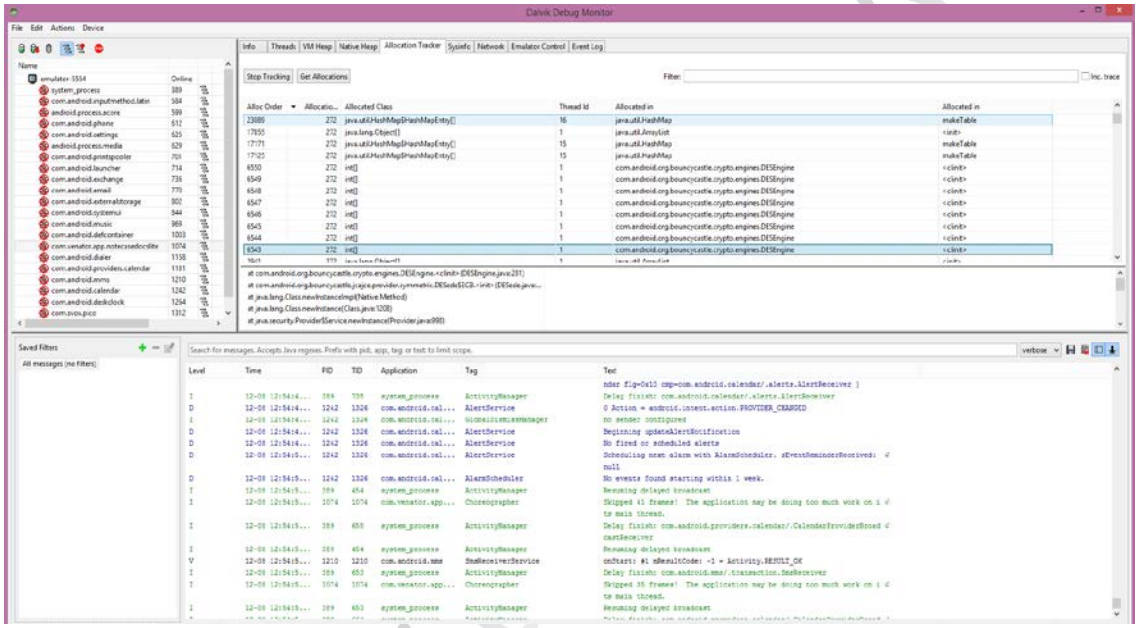

Figure 31: Track Memory Allocation using DDMS

7. Results & Evaluation

The exact results produced by our applications' analyses are cited in the current chapter. The majority of the applications were examined applying both static and dynamic analyses. Due to specific particularities of some applications however, there are certain cases where the applications were examined exclusively by employing static analysis, the results of which are also included in our work and presented in table 2, while the findings of the former applications are presented in the table 1.

This chapter also includes the results of the applications testing scenarios discussed in chapter 4. These particular results -presented in table 3- concern the total amount of the applications examined and specify the applications that are not IND-CPA secure, based on their output. Combining both types of results, we are able to form a more accurate opinion regarding Android applications security.

	Android Application	**Category**	**Vulnerabilities (Static Analysis)**	**Vulnerabilities (Dynamic Analysis)**	**Rules Violated**
1	AT&T Secure Messaging	*Secure Messaging*	• Key Generation: RSA/**ECB/PKCS1 Padding** • Message Digest: **MD5**	• android.AndroidDigestFactory.**getSha1**	• *Rule 1* • *Rule 6*
2	BlackSms Text Crypt	*Secure Messaging*	• **Not known algorithm usage for encryption.** • **Hard coded and constant salt**.		• *Rule 2* • *Rule 16*
3	Crypt	*Secure Messaging*	• Secret Key Factory: PBKDF2WithHmac**SHA1**	• PBKDF2WithHmac**SHA1** • Digest: **getSHA1** • PBEKeySpec: PBKDFf2withHmac**SHA1**	• *Rule 1*
4	Crypt Haze	*Secure Messaging*	• Password Generation: **MD5** • **Hard coded and constant salt.** • **PBE used with 20 iterations.**		• *Rule 1* • *Rule 16* • *Rule 17*
5	CryptoDroid	*Secure Messaging*	• **Usage of Caesar Cipher and Columnar Transposition algorithms for encryption.**		• *Rule 1*

6	Encrypt and Send	*Secure Messaging*	**Not known algorithm usage for encryption.**		• *Rule 2*
7	Encrypt It	*Multipurpose Encryption Utilities*	**AES call without specific mode defined.**	• bouncycastle.crypto.engines.AESFastEngine: **AES$ECB**	• *Rule 5*
8	Encrypted Notepad	*Document Encryption*	Message Digest: **MD5, SHA1** AES-**CBC-PKCS5Padding**	No further results.	• *Rule 1* • *Rule 24*
9	FREE Secure Message Encryption	*Secure Messaging*	**Not known algorithm usage for encryption.**		• *Rule 2*
10	Gryphn Secure Text Messaging	*Secure Messaging*	RSA/**ECB**/PKCS1Padding		• *Rule 6*
11	Keep A Secret	*Secure Messaging*	**MD2/MD5/SHA1/ Manual IV's creation PBE with hard coded and constant salt.** AES-**CBC-PKCS5Padding**	• bouncycastle.jcajce.provider.symmetric.AES$PBEWith**SHA**And256BitAESBC	• *Rule 1* • *Rule 7* • *Rule 16* • *Rule 24*
12	LmxSecure Free	*Secure Messaging*	AES-256/**CTR**/PKCS5Padding Message Digest: **MD5** **Hard coded IV**. **CTR with constant IV**.		• *Rule 1* • *Rule 8* • *Rule 12* • *Rule 13*
13	myENIGMA Secure Messaging	*Secure Messaging*	AES/**ECB**/NoPadding Message Digest: **SHA1**, **MD5** **CBC-CTR Implementations**		• *Rule 1* • *Rule 3* • *Rule 4*
14	SafeSlinger	*Sensitive Data Exchange & Storage*	Key Testing: AES/**ECB**/NoPadding **AES call without specific mode defined.**	• com.android.org.bouncycastle.jcajce.provider.symmetric.**AES$ECB**$1	• *Rule 4* • *Rule 5*
15	Secret Coder Lite	*Secure Messaging*	• Usage of AES/**ECB**/PKCS7 Padding, **Morse Code, Letter to Number, Letter to Symbol, Reversed Alphabet, Backwards and Caesar Shift for encryption.** • Message Digest: **MD5** • AES-**CBC-PKCS5Paddinf** • **Hard-coded key.**	AES available to pro version	• *Rule 1* • *Rule 4* • *Rule 10* • *Rule 11* • *Rule 24*

16	Secret SMS	*Secure Messaging*	• AES-**CBC-PKCS5Padding**	No further results.	• *Rule 24*
17	SMS Crypt	*Secure Messaging*	No cryptography found.	No further results.	
18	mSecure	*Sensitive Data Exchange & Storage*	• Signature: **SHA1**WithRSA • **Blowfish**	No further results.	• *Rule 1*
19	Passdroid	*Sensitive Data Exchange & Storage*	• **Hard coded key.**	• com.android.org.bouncycastle.jcajce.provider.symmetric.AES$**ECB**$1	• *Rule 4* • *Rule 10* • *Rule 11* •
20	Encrypt Notes	*Document Encryption*	• **AES called without specific mode defined.**	• com.android.org.bouncycastle.jcajce.provider.symmetric.AES$**ECB**$1	• *Rule 4* • *Rule 5*
21	aVault Secure Document Encrypt	*Document Encryption*	• LocalMessageDigest: **SHA1** • **PBE with 256 iterations.** • AES-**CBC-PKCS5Padding**	• JCEBlockCipher	• *Rule 1* • *Rule 17* • *Rule 24*
22	URSafe Notecase	*Document Encryption*	• **AES called without specific mode defined.** • **DESede** • Key Creation: **Blowfish** • **Statically defined salt.** • **PBE with fewer than 1000 iterations.**	• com.android.org.bouncycastle.cryptoengines.**DES** Engine • Javax.crypto.spec.**DESedeKeySpec** • Jcajce.provider.symmetric.**DESede$ECB**	• *Rule 1* • *Rule 4* • *Rule 5* • *Rule 9* • *Rule 16* • *Rule 17*
23	CryptoNotes	*Document Encryption*	• **AES called without specific mode defined**. • **Hard coded key.** • **Improper use of dofinal().**	• com.android.org.bouncycastle.jcajce.provider.symmetric.AES$**ECB**$1	• *Rule 4* • *Rule 5* • *Rule 10* • *Rule 11* • *Rule 22*

24	Secret Box	*Multipurpose Encryption Utilities*	• **AES called without specific mode defined.** • Digest: **MD5** & **SHA1** (apache) • **PBE with 128 iterations.**	• JCEBlockCipher • AESFastEngine	• *Rule 1* • *Rule 5* • *Rule 17*
25	SafeNote2	*Document Encryption*	No cryptography found.	No further results.	
26	Secret Message (unknown)	*Secure Messaging*	• **MD5** • AES-**ECB**-PKCS5Padding • **Non-securely random seed for key creation.** • AES-**CBC-PKCS5Padding**	• JCEBlockCipher • AES-**ECB**	• *Rule 1* • *Rule 4* • *Rule 20* • *Rule 21* • *Rule 24*
27	Secret Message Elite	*Secure Messaging*	• **PBEWITHMD5ANDDES** • **Hard coded salts.** • **Non-securely random seed for IV.** • **PBE with 11 iterations.** • **Non-securely random seed for key creation.** • **AES Implementation** • AES-**CBC-PKCS5Padding**	• AES-**ECB**	• *Rule 1* • *Rule 3* • *Rule 4* • *Rule 7* • *Rule 16* • *Rule 17* • *Rule 19* • *Rule 20* • *Rule 21* • *Rule 24*
28	Secret Messages CryptApp	*Secure Messaging*	• **MD2** • **MD5** • **SHA-1** • **DES**-CBC-NoPadding • **Blowfish**-CBC-NoPadding • **Hard-coded seed for IV's creation.**	• get**SHA1**	• *Rule 1* • *Rule 9* • *Rule 19*
29	Encrypt File Free Application	*Document Encryption*	• **MD5** • **Not known algorithm usage for encryption.**		• *Rule 1* • *Rule 2*

30	SSE	*Multipurpose Encryption Utilities*	• **MD5** • **Blowfish**-256, **Blowfish**-448, **Gost-256, Serpent-256, Twofish, RC6, CRC32** • **SHA-1** • **Non-securely random IV created.**	• com.android.org.conscrypt.OpenSSLMessageDigestJDK (**getMD5Hash**) • sse.utils.Encryptor (**getMD5Hash**) Bouncycastle: • crypto.enginges.**BlowfishEngine**: getBaseCBCCipher • crypto.digests.**SHA1Digest** • crypto.prng.ThreadedSeedGenerator (**generateSeed**)	• *Rule 1* • *Rule 9* • *Rule 15* • *Rule 20*
31	Cipher Cryptics (Tool Cipher)	*Multipurpose Encryption Utilities*	• **ASCII Cipher, AtBash Cipher, Bifid Cipher, Binary Cipher, Caesarian Shift Cipher, Enigma Cipher, Hex Cipher, Oct Cipher, Playfair Cipher, ROT13 Cipher, Vigenere Cipher**		• *Rule 1*
32	Decipher – Code and Cipher Kit (Cipher Project)	*Multipurpose Encryption Utilities*	• **AtBash, Binary, Octal, Hexadecimal, Caesar Shift, Keyword, Substitution, Vigenere, Beaufort, Hill Cipher** • **MD5**, **SHA-1**		• *Rule 1*
33	Secrecy – Encrypt/ Hide Files	*Document Encryption*	• AES-**ECB**-PKCS5Padding • **SHA-1** • AES-**ECB**-PKCS7Padding	No further results.	• *Rule 1*

34	**Crypto Tool DUKPT**	***Sensitive Data Exchange & Storage***	• **DESede-ECB-NoPadding** • **DESede-CBC-NoPadding** • **DES-ECB-NoPadding**	**No further results.**	• *Rule 1* • *Rule 4* • *Rule 9*
35	Crypto (Burrows)	*Multipurpose Encryption Utilities*	• **ALDER32, BASE64, CRC32, Binary, Decimal, Hexadecimal, MD4, MD5, RIPEMD160**, **SHA-1**		• *Rule 1*
36	Secure Notes (inno)	*Document Encryption*	• **DES/CBC/PKCS5Padding** • **DESKeySpec** • Signature: **SHA1**WithRSA (trivialdrivesample) • **Hard coded IV.**	No further results.	• *Rule 1* • *Rule 9* • *Rule 12* • *Rule 13* • *Rule 24*
37	Secure Notes (anpo)	*Document Encryption*	• PBEKeySpec: **PBEWithMD5AndDES** • **Hard coded password used in PBE.** • **Hard coded salt in PBE.** • **PBE with 19 iterations.**	• com.android.org.bouncycastle.crypto.engines.**DES Engine: DES$PBEWithMD5**	• *Rule 1* • *Rule 16* • *Rule 17* • *Rule 18*
38	Secure Note	*Document Encryption*	No cryptography found.	No further results.	
39	Encrypt Bancomat	*Sensitive Data Exchange & Storage*	No cryptography found.	No further results.	
40	Secure Notepad	*Document Encryption*	No cryptography found.	No further results.	

Table 3: Results of Static and Dynamic Analyses

	Android Application	Category	Vulnerabilities (Static Analysis)	Rules Violated
1	Cipherize, Encryption Tool	*Multipurpose Encryption Utilities*	• AES, **Blowfish, DES, DESede**, Twofish, **Vigenere, Alphanumeric, Morse, ASCII, AtBash, Decallage, BatonsRomain, CarreDePolybe, Pigpen, MD5,** WehrmachtEnigma • **PBE with hard coded and constant salt.**	• *Rule 1* • *Rule 9* • *Rule 16*
2	Crypt 2 Share	*Document Encryption*	• **AES call without specific mode defined.** • **Key creation by statically assigning values in a bytearray.**	• *Rule 5* • *Rule 11*
3	Crypt Text Messages	*Secure Messaging*	• **Blowfish**: CBC-**ECB**-CFB implementations • Message Digest: **SHA1** • **Hard coded key used for encryption**	• *Rule 1* • *Rule 3* • *Rule 4* • *Rule 9* • *Rule 10*
4	Encrypt Email	*Secure Messaging*	No cryptography found	
5	Startel Secure Messaging	*Secure Messaging*	• **PBEWithMD5AndDES** • **PBE with 20 iterations.** • **Hard coded and constant password in PBE.**	• *Rule 1* • *Rule 16* • *Rule 18*
6	Text Secure Beta	*Secure Messaging*	• AES-CTR-NoPadding (Session Cipher) • Keys: **SHA-1** • AES-**CBC-PKCS5Padding**	• *Rule 1* • *Rule 7* • *Rule 8* • *Rule 24*
7	Cyphr	*Secure Messaging*	• **SHA-1** (crashlytics) • AES-**ECB**-PKCS7Padding (crashlytics)	• *Rule 1* • *Rule 4*
8	Surespot	*Secure Messaging*	• **DES/ECB/**NoPadding • **MD5**	• *Rule 1* • *Rule 4*
9	Telegram	*Secure Messaging*	• **MD5** • **Insecure IV's creation**	• *Rule 1* • *Rule 15* • *Rule 20*

Table 4: Results of Static Analysis

Regarding the content of the aforementioned and already presented results, although the majority of the applications use AES in CBC mode, there is a significant number of android source codes that include either ECB mode, which as discussed before should be strictly avoided, or at least one obsolete algorithm. To be more specific, the vast majority of android applications' source codes encompass at least one cryptographic misuse, not always relevant to cryptographic algorithm and mode selection. Nonetheless, there are many cases where no cryptography is detected or out of date algorithms are invoked, for instance Caesar Cipher, Columnar Transposition, AtBash Cipher and Playfair Cipher along with others, even by applications bearing a name that implies the use of strong cryptography.

It is noted that for those specific cases of applications employing algorithms similar to Caesar's Cipher or algorithms implemented by the developers, it is expected for its dynamic analysis to produce no results, as when an unofficial cryptography solution is used then their employment cannot be detected via android memory usage. Furthermore, the same applies to the cases where RSA is used with ECB mode, as in the specific scheme, RSA is invoked but ECB is not finally called.

As far as a more specific and statistical analysis of the results is concerned, it seems that the applications violating at least one rule –taking into consideration the findings of both static and dynamic analyses- reach a percentage of 87, 75% (i.e. 43 out of 49 applications) while the applications in which no cryptography detected to be employed reach the 12, 24% (i.e. 6 out of 49 applications). Consequently, the percentage of the applications that seem to violate none of our defined rules is approximately 0% (i.e. none of the 49 applications).

Furthermore, after having completed our research, we deduced that 71, 42% of the tested applications (i.e. 35 out of 49 applications) use either a broken or a weak cryptographic algorithm (rules 1, 2 and 3). Simultaneously, the percentage of the applications employing cryptographic techniques with incorrect parameters (rules 7, 8, 16, 17, 18) reach the 22, 44% of the applications examined (i.e. 11 out of 49 applications). Last but not least, the applications combining CBC mode with PKCS5Padding (rule 24) reach the 16, 32% (8 out of 49 applications).

• **Applications violating at least one rule**	• **Applications where no Cryptography was detected**	• **Applications where no rule violation was detected**	• **Broken or weak cryptographic algorithm Usage**	• **Incorrect cryptographic parameters employment**	• **Applications Combining CBC with PKCS5Padding**
• **87, 75%**	• **12, 24%**	• **0%**	• **71, 42%**	• **22, 44%**	• **16, 32%**

Table 5: Cryptographic misuse percentages in Android Applications

As for the results of our experimental scenarios of chapter 4, we deduced that leastwise the 26, 53% of the applications (i.e. 13 out of 49 applications) are not IND-CPA secure (table 6).

No	Non IND-CPA Secure Application	Finding
1	**BlackSmS Text Crypt**	• Scenario 1 not Satisfied
2	**Crypt**	• Scenario 1 not Satisfied
3	**Crypt Haze**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
4	**CryptoDroid**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied
5	**FREE Secure Message Encryption**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
6	**LmxSecure Free**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
7	**Secret Coder Lite**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
8	**Secret Message (unknown)**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
9	**Secret Message Elite**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
10	**Secret Messages CryptApp**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
11	**Crypto (Burrows)**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
12	**Encrypt Bancomat**	• Scenario 1 not Satisfied

		• Scenario 2 not Satisfied • Scenario 3 not Satisfied
13	**Cipherize, Encryption Tool**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
14	**Cipher Cryptics**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied
15	**Decipher**	• Scenario 1 not Satisfied • Scenario 2 not Satisfied • Scenario 3 not Satisfied

Table 6: Non IND-CPA secure applications

No	Android Application with Native Code
1	**Crypt Text Messages**
2	**CryptoDroid**
3	**Secure Notes (anpo)**
4	**Crypto (burrows)**
5	**Secure Notes (inno)**
6	**Secure SMS**
7	**Telegram**
8	**myENIGMA**
9	**Gryphn**
10	**SSE**
11	**Encrypt Email**

Table 7: Android applications with native code

8. Conclusions

In this current thesis, after having thoroughly studied the basic principles of cryptography, we did examine the tendencies of Android developers as far the cryptography employment is concerned.

It is fact that relevant research has been conducted in the specific field, showing that developers tend to neglect basic cryptographic principles, as the results show that the majority of android applications use either obsolete cryptography or improper encryption parameters. However, related works, always based on results of automated tools, have not yet provided specific and accurate results regarding cryptographic misuses of the applications. What is more, developers' community continues to lack a specifically defined list of cryptographic misuses and best practices.

Taking this fact into consideration, we concentrated a sufficient number of cryptographic rules that will help android developers who are not information security specialists –and not only- protect sensitive data in android applications.

Subsequently, depending our work on the aforementioned list of cryptographic rules, we tested -using static and dynamic analysis techniques- 49 Android applications, downloaded between June and November 2014.

Our results confirmed the results of related work, showing that 87, 75% violate at least one cryptographic rule and that the 26, 53% of the applications tested (at minimum) are not IND-CPA secure. Thus, we can deduce that a significant percentage of android developers lack knowledge of fundamental cryptographic principles and techniques, which are not yet consolidated in developers' community. For this reason, the list proposed by this thesis could be used to help programmers initiate themselves in cryptography's basics.

References

[1] recombu, "*What is Android and what is an Android phone?*", http://recombu.com/mobile/article/what-is-android-and-what-is-an-android-phone_M12615.html

[2] Heavy, "*What is Android? Top 10 facts You Need to Know*", http://heavy.com/tech/2013/06/what-is-android-os-operating-system-info-wiki/

[3] Maximumpc, "*MobileOS Revolution: Android 4, Windows Phone 7.5, and Apple iOS 5 Explained*", http://www.maximumpc.com/article/features/mobile_os_revolution_android_4_windows_phone_75_and_apple_ios_5_explained

[4] Wikipedia, "*Android (operating system)*", http://en.wikipedia.org/wiki/Android_%28operating_system%29

[5] Wikipedia, "*Android Version History*", http://en.wikipedia.org/wiki/Android_version_history

[6] androidcentral, "*Inside the different Android Versions*", http://www.androidcentral.com/android-versions

[7] Brahler S., Analysis of the Android Architecture, http://os.itec.kit.edu/downloads/sa_2010_braehler-stefan_android-architecture.pdf

[8] Slideshare, Android Activity Lifecycle,

http://www.google.gr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=0CAUQjhw&url=http%3A%2F%2Fwww.slideshare.net%2FSohamPatelPatel%2Fandroid-activity-lifecycle-39195427&ei=lV-AVJuEBtLoaLC3gLAH&bvm=bv.80642063,d.ZGU&psig=AFQjCNG7JILgvklmlCiIrVrkDwd3aSlPPw&ust=1417785548838541

[9] bestandroidtraining, Android Project Structure and Activity Lifecycle,

https://www.google.gr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRw&url=https%3A%2F%2Fbestandroidtraining.wordpress.com%2F2013%2F03%2F30%2Fandroid-project-structure%2F&ei=AWiAVIDUIMvraOGGgMAB&bvm=bv.80642063,d.ZGU&psig=AFQjCNEXC2hRGzY2-vVNjZsowXGyEFfg5Q&ust=1417787712199471

[10]C. Enrique Ortiz, http://CEnriqueOrtiz.com

[11] hiqes, "*Android Security Part I: App Basics*", http://hiqes.com/android-security-part-1/

[12] BATYUK, Leonid ; SCHMIDT, Aubrey-Derrick ; SCHMIDT, Hans-Gunther; CAMTEPE, Ahmet ; ALBAYRAK, Sahin: Developing and Benchmarking Native Linux Applications on Android. In: MobileWireless Middleware, Operating Systems, and Applications, 2009, 381–392

[13] Shahzad M., "*Native methods in Java – Java Native Interface (JNI)*", http://mudassirshahzad.com/native-methods-in-java-java-native-interface-jni/

[14] The Java™ tutorial, "*Lesson: Overview of the JNI*", http://www.math.uni-hamburg.de/doc/java/tutorial/native1.1/concepts/index.html

[15] tutorialspoint, "*Android Architecture*", ,http://www.tutorialspoint.com/android/android_architecture.htm

[16] Scriptol, "*Dalvik, the virtual machine of Android*", http://www.scriptol.com/programming/dalvik.php

[17] http://coltf.blogspot.gr/p/android-os-processes-and-zygote.html

[18] "*What Android Is*", http://www.tbray.org/ongoing/When/201x/2010/11/14/-big/Framework.png.html

[19] "*Android 5.0 Data Better Protected With New Crypto System*", http://blog.kaspersky.com/full-disk-encryption-android-5/

[20] Introduction to Cryptography, http://vig.prenhall.com/samplechapter/0130614661.pdf

[21] Bellare, Rogaway, Symmetric Encryption, https://cseweb.ucsd.edu/~mihir/cse207/w-se.pdf

[22] Bellare, Rogaway, Block Ciphers, ,http://cseweb.ucsd.edu/~mihir/cse107/w-bc.pdf

[23] Canteaut A., Stream Cipher, https://www.rocq.inria.fr/secret/Anne.Canteaut/encyclopedia.pdf

[24] Bellare, Rogaway, Asymmetric Encryption, https://cseweb.ucsd.edu/~mihir/cse207/w-asym.pdf

[25] Manuel Egele, David Brumley: Carnegie Mellon University, Yanick Fratantonio, Christopher Kruegel, University of California, Santa Barbara, An Empirical Study of Cryptographic Misuse in Android Applications

[26] OWASP, Static Code Analysis, https://www.owasp.org/index.php/Static_Code_Analysis#False_Positives

[27] Yong Li, Yuanyuan Zhang, Juanru Li, Dawu Gu, iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications, Dept. of Computer Science and Engineering Shanghai Jiao Tong University Shanghai, China

[28] Somak Das, Vineet Gopal, Kevin King, Amruth Venkatraman, IV = 0 Security Cryptographic Misuse of Libraries, 6.857 Final Project, May 14, 2014

[29] Nishika, Rahul Kumar Yadav, Cryptography on Android Message Applications – A Review, PDM College of Engineering Bahadurgarh, India

[30] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, Shi Chenjie, Modeling Analysis and Auto Detection of Cryptographic Misuse in Android Applications, China Information Technology Security Evaluation Center, Beijing University of posts and telecommunications, IEEE, 2014

[31] The MITRE Corporation, Using Automated Static Analysis Tools for Code Reviews, http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/using-automated-static-analysis-tools-for, 2013

[32] Nigel Smart, Advances in Cryptology – EUROCRYPT 2008, Springer

[33] Javamex, Password Based Encryption in Java, http://www.javamex.com/tutorials/cryptography/password_based_encryption.shtml

[34] David Lazar, Haogang Chen, Xi Wang, Nickolai Zeldovich, Why does cryptographic software fail? A case study and open problems, MIT CSAIL

[35] Android Developers, Security Tips, http://developer.android.com/training/articles/security-tips.html

[36] Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 7, http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html

[37] Java Cryptography Architecture (JCA) Reference Guide, http://docs.oracle.com/javase/8

[38] http://armoredbarista.blogspot.gr/2012/09/rsaecb-how-block-operation-modes-and.html

[39] NIST, NIST Cryptographic Standards and Guidelines Development Process, Report and Recommendations of the Visiting Committee on Advanced Technology of the National Institute of Standards and Technology, July 2014, http://www.nist.gov/public_affairs/releases/upload/VCAT-Report-on-NIST-Cryptographic-Standards-and-Guidelines-Process.pdf

[40] Wikipedia, Key Generation, http://en.wikipedia.org/wiki/Key_generation

[41] "Best Practices for encryption in Android", http://afu.com/blog/MFE_Encryption_WhitePaper_v1.pdf

[42] IT&C, Solutions and tutorials for IT&C development, How to encrypt/decrypt files in Java with AES in CBC mode using Bouncy Castle API and NetBeans or Eclipse, http://www.itcsolutions.eu/2011/08/24/how-to-encrypt-decrypt-files-in-java-with-aes-in-cbc-mode-using-bouncy-castle-api-and-netbeans-or-eclipse/

[43] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo, Simionato, Graham Steel, Joe-Kai Tsay, Efficient Padding Oracle Attacks on Cryptographic Hardware, HAL, last revised 25 Jan 2012

[44] Robert Heaton, The Padding Oracle Attack – why crypto is terrifying, http://robertheaton.com/2013/07/29/padding-oracle-attack/, July 29, 2013

[45] SkullSecurity, Padding Oracle Attacks: in depth, https://blog.skullsecurity.org/2013/padding-oracle-attacks-in-depth

[46] John's Cryptography Blog, AES CBC Padding Oracle Attack, http://johnx.blogspot.com/2010/10/aes-cbc-padding-oracle.html, October 21, 2010

[47] Juliano Rizzo, Thai Duong, Practical Padding Oracle Attacks, May 25th, 2010

[48] Serge Vaudenay, Security Flaws Induced by CBC Padding – Applications to SSL, IPSEC, WTLS …, Swiss Federal Institute of Technology (EPFL)

[49] Vlastimil Klima, Tomas Rosa, Side Channel Attacks on CBC Encrypted Messages in the PKCS#7 Format

[50] TechTarget, "*Static Analysis* ", http://searchwindevelopment.techtarget.com/definition/static-analysis

[51] viva64, "*Static Code Analysis*", ,http://www.viva64.com/en/t/0046/

[52] gcn, "*Static vs. Dynamic Code Analysis: advantages and disadvantages*", http://gcn.com/articles/2009/02/09/static-vs-dynamic-code-analysis.aspx

[53] veracode, "*Static Testing vs. Dynamic Testing*", https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing

[54] intel, "Dynamic Analysis vs. Static Analysis", https://software.intel.com/sites/products/documentation/doclib/iss/2013/inspector/lin/ug_docs/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm

[55] Oracle, Java SE Documentation, Java Cryptography Architecture (JCA) Reference Guide,

http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

[56] Intel, Developer Zone, Sample Code: Data Encryption Application, https://software.intel.com/en-us/android/articles/sample-code-data-encryption-application

[57] Android Developers Blog, Using Cryptography to Store Credentials Safely, http://android-developers.blogspot.gr/2013/02/using-cryptography-to-store-credentials.html

[58] Code 2 learn, Java: Encryption and Decryption of Data using AES algorithm with example code, http://www.code2learn.com/2011/06/encryption-and-decryption-of-data-using.html