Radboud University Nijmegen

TRU/e Security
Faculty Of Science
Radboud University, Nijmegen, Netherlands

Cryptographic (In)Security in Android Apps

An Empirical Analysis

Master Thesis in Cyber Security

Author:
Milad Masoodi

Supervisor:
Dr. Veelasha Moonsamy

Second reader:
Dr. Fabian van den Broek

July 20, 2020

As our circle of knowledge expands, so does the circumference of darkness surrounding it.

—Albert Einstein

Acknowledgements

No work of immense effort has ever been accomplished by the trials of one man alone, as such is this humble attempt. While I have performed this work through great dedication and effort relative to my situation, I could never achieve a final result of such adequacy without the help of many adept and selfless scientists, instructors, and friends that aided me in every step of the way.

Atop all is Dr. Moonsamy; her excellent guidance and insight in critical moments on intricate details shaped this work. Along with the staff of the cybersecurity group of Radboud University, exceptional scientists that opened the gates of this fantastic field for me, and not to forget the invisible staff of the department, those who clean, maintain, and keep the environment operational so we could learn.

It behooves me to thank Dr. Manuel Egele for CryptoLint, the incredible software that helped me design mine, my wonderful uncle, friend, and mentor Kamal Mohajerani who provided help, support, guidance, and elucidated every consultation with care and patience which considerably affected the format of this research, and my very good friend Sina Afshar for proofreading and commenting on my thesis.

Last but not least, my mother, Dr. Sue Mohajerani, whose grand sacrifice made all of this ever possible.

Abstract

For so long Android developers have relied on third-party documentation for their cryptographic needs; this, however, has changed in recent years with the gradual introduction of Android's documentation. Nevertheless, the accuracy of the documentation and their effect on the cryptographic security of Android apps in the wild has not been well-studied so far.

This thesis aimed to shed light on these topics with several approaches; we defined a set of rules that we perceive to represent cryptographic security, then we thoroughly scrutinized the documentation around those rules, after that we designed and executed series of automated tests to analyze 6900 real apps to grasp an understanding of the current state of their security.

Eventually, we concluded that Android documentation not only fails to contribute to the cryptographic security of the apps, but it also has an adverse effect on the matter as with more time, number of insecure apps in the circulations have gradually increased.

Contents

List of Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
APK	Android Package File
ART	Android Runtime
CBC	Cipher Block Channing
CFG	Control Flow Graph
CG	Call Graph
CHF	Cryptographic Hash Function
CIL	Common Intermediate Language
CMA	Cryptographic Misuse Analyzer
CSV	Comma Separated Values
CTR	Counter Mode
DEX	Dalvic Executable
ECB	Electronic Code Book
GCM	Galois/Counter Mode
GUI	Graphic User Interface
HKDF	Hash based Key Derivation Function
ID	Identifier
IND-CCA	Indistinguishability under Chosen Ciphertext Attacks
IND-CPA	Indistinguishability under Chosen Plaintext Attacks
IPC	Inter-Process Communication
IV	Initialization Vector
JCA	Java Cryptography Architecture
JCP	Java Cryptographic Provider
KDF	Key Derivation Functions
MAC	Message Authentication Code

NIST	National Institute of Standards and Technology
OS	Operating System
PBE	Password-Based Encryption
PRNG	Pseudo Random Number Generator
RegEx	Regular Expressions
RNG	Random Number Generator
SDK	Software Development Kit
SHA1PRNG	Secure Hash Algorithm 1 Random Number Generator
TRNG	True Random Number Generator
VM	Virtual Machine

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The origins of "smartphones" can be traced back to 1994 with "IBM Simon Personal Communicator" or simply "IBM Simon" [1]. Over the years, the technology has gone through multiple evolution cycles and many generations. The modern iteration of smartphones that are currently being used, was introduced in 2007 with the first iPhone, followed by the emergence of Android phones. Since then the smart devices industry has revolutionized our lives and continue to do so till date.

For many years, Android has been a dominant player in the industry, with a growing presence since its inception. With staggering figures of more than 80% of the market share [2] and 2, 887, 841 [3] apps in its official store, GooglePlay[1], it seems that in the foreseeable future, Android will remain to play big role in the smartphone market. In every minute of 2018 alone, 456,000 tweets, 46,740 Instagram posts, 16 million text messages, 990,000 Tinder swipes had been processed, 80% of which was done with Android devices [4].

This unprecedented presence of a device in which we have little control over has risen many concerns and problems regarding its privacy and security of users' sensitive data [5]. Albeit, malware analysis of Android apps is indeed a vast and active area of research, many security problems in the Android landscape stem in how apps are developed as many legitimate developers fall victim to the pitfalls of development complexity. This is particularly true for the field of cryptography, while immensely intricate, with many details that can go wrong, it is indeed very complicated for everyday developers to fully comprehend. This complexity can cause developers to rely on external sources and gratuitously replicate code provided by others to provide security for their apps.

Finding the correct reference is a constant struggle for all programmers, many of whom rely on publicly available forums such as "StackOverflow"[2] for their everyday needs. Although this is a generally acceptable and reliable solution for most programming problems, it is not as such for cryptographic issues. Mainly because in cryptography, many concepts are abstract and incorrect implementation cannot be immediately observed by programmers. While in other disciplines, if a code is wrong, it is immediately noticeable during development or testing. In case of cryptography to guarantee that the implementation is correct the solution is to always seek answers from official sources, in this case, from Android documentation website[3]. Thus, availability, correctness, and

[1] https://play.google.com
[2] https://stackoverflow.com
[3] https://developer.android.com

completeness of the documentation is of critical importance.

Consequently, it is paramount to ensure that the developers do indeed follow correct cryptographic principles during the development process. Prior studies have looked into the cryptography in Android apps. Notably, in a 2013 paper by Egele et al. [6] it is shown that many apps in Android's PlayStore suffered from some cryptographic misuse. However, back in 2013, the primary source of information for Android developers was third-party forums such as StackOverflow. Yet, the situation has been changed. Android documentation has enormously evolved to be much more comprehensive, and include many more guidelines and best practices in contrast with the past.

1.1 Objectives

To understand if Android documentation is having an advantageous effect on its apps, the following questions must be answered:

- How comprehensive, usable, and reliable is the Android documentation regarding implementation of cryptographic libraries in Android apps?

  The *comprehensiveness* of the documentation is essential since it demonstrates the diversity of the problems that it can cater. *Usability* would present an indication of how easy or hard it is to solve problems developers may encounter. *Reliability* would express the correctness of the documentation, and if they are thoroughly followed, will it ensure a secure result?

- How effective have the changes in the documentation been on cryptographic correctness of apps in the Android PlayStore?

Finding the answer to these questions will help grasp a better understanding of security and reliability of android apps in general, detailing:

1. If the creators of the system (Google) indeed achieved their goals of securing the Android app ecosystem;
2. If the documentation is done properly;
3. How reliable are Android apps?
4. Is Android on the right track on its documentation provision?

Rationally, after an in-depth investigation into the problem, we would have a good understanding of the structure and quality of the documentation. After experimentation, we will have a quantifiable measure of what percentage of apps in the Android market are appropriately implemented. Hence, the effectiveness of the documentation will be clarified. Therefore, if there is evidence that the documentation is well done and relative to prior research, a smaller percent of apps will suffer from cryptographic misuse. We can indeed confidently hypothesize that the changes have been useful, meaning that Google is on the right path to creating a secure software ecosystem.

1.2 Procedure

The logical steps to approach the problem would be to first conduct a comprehensive study into the background of the issue, such as its practical and cryptographic aspects. This study will aid us in fully understanding the problem. In this step, we decide on the criterion in which a proper app and a faulty one are distinguished. Next, we continue to an in-depth study of Android cryptography concerning documentation to evaluate their readability and quality. Subsequently, we move on to the analysis of real-world apps developed in the past couple of years. This experiment will help us compare the earlier research and draw a conclusion on the progress in recent years.

1.3 Structure

To deliver the research clearly, we must first conduct an in-depth background study, both from the Android operating system and cryptographic point of view. To be able to do so, we present two background chapters for each aspect of the problem. First, Chapter 2 focuses on the study of Android as a whole, Section 2.1 looks into the Android operating system, versions, compatibility, and operation system (OS) security, Then in Section 2.2, we limit our focus on Android apps, their structure, and reverse engineering techniques, Lastly, in Section 2.3, we discuss the source of the documentation. Second, Chapter 3 moves on to the cryptographic aspects of the problem. In Section 3.1, we define the cryptographic properties that we aim to evaluate the apps with, and we further elaborate and bring forward the reasons for the feasibility of this decision. Later, Section 3.2 defines the set of rules based on which the apps will be evaluated, and then we carefully reason and clarify each of the rules. Next, in Chapter 4, we look into the existing work by previous scholars and the community alike, and we define the knowledge base and the work that has been done so far in both cryptographic analysis of the apps and Android reverse engineering. Afterward, in Chapter 5 we look into each of the rules and investigate how they are represented by Android documentation. This chapter will help to answer the first research question. Later, in Chapter 6, we design and conduct an experiment to determine the effect of the documentations on the app ecosystem. Subsequently, we methodologically analyze the result of the experiment in Chapter 7, first in Section 7.1 with an evolutionary approach and later in Section 7.2 we look into the latest state of cryptographic security in apps. We represent the findings in Section 7.3. Lastly, in the final chapters, we present the answers to the research questions, present recommendations, suggest future work, and conclude the research.

Chapter 2

Android Preliminaries

This chapter introduces the fundamental knowledge base required to establish this thesis. Indeed, each of the topics discussed in this chapter and the following can be inspected to a great extent, yet, it is intended to keep the information as concise and relevant as possible.

First, in Section 2.1, a general overview of the Android operating system is presented, explaining its origins and infrastructure, with a focus on the general security of the system. Section 2.2 discusses the inner structure of Android apps is discussed to introduce the reader to its components. Introduction about Android app decompilation, information extraction, and methods to access and modify them is presented. Next, in Section 2.3, ways to access Android developer guidelines are investigated.

2.1 Android OS

Android OS is an open-source mobile operating system developed based on Linux kernel with specific design, oriented around touch-screen devices. Developed by Open Handset Alliance[1], Android Inc., and owned and maintained by Google Inc.

Android OS is designed with versatility in mind, so that it can be installed on a variety of hardware, from all manufacturers. In Android, apps run on a virtual machine platform called *Android Runtime (ART)*, allowing every app to run effectively in a separate process, with its instance of the Virtual Machine (VM). ART has been designed to execute multiple VMs efficiently.

Apps benefit from this approach too. An app source code is compiled into specific bytecodes, called `.dex` files or Dalvik Executable Files (a remnant of Dalvik Virtual Machine [7], which was the earlier version of the runtime environment) that contain all of the apps' classes and assets. `.dex` files are designed for optimized memory management and data sharing capabilities [8].

2.1.1 Android Version History

Since its inception in 2003 and acquisition by Google in 2005, Android has gone into 18 releases and ten versions. Table 2.1 presents Android version history [9] in detail accompanied by the subsidiary information in regards to each release.

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

Each version of Android is assigned a codename that is a string representing its version for ease of addressing, a version number, which is a double-precision number indicating the detail of the update, an API (Application Programming Interface) level [10], which is an integer value used for identifying the framework API revision offered by a version of the Android platform.

Table 2.1: Android version history

Codename	**Version**	**Release date**	**API level**
No codename	1.0	Q3-2008	1
No codename	1.1	Q1-2009	2
Cupcake	1.5	Q2-2009	3
Donut	1.6	Q3-2009	4
Eclair	2.0 – 2.1	Q4-2009	5 – 7
Froyo	2.2 – 2.2.3	Q2-2010	8
Gingerbread	2.3 – 2.3.7	Q4-2010	9 – 10
Honeycomb	3.0 – 3.2.6	Q1-2011	11 – 13
Ice Cream Sandwich	4.0 – 4.0.4	Q4-2011	14 – 15
Jelly Bean	4.1 – 4.3.1	Q3-2012	16 – 18
KitKat	4.4 – 4.4.4	Q4-2013	19 – 20
Lollipop	5.0 – 5.1.1	Q4-2014	21 – 22
Marshmallow	6.0 – 6.0.1	Q4-2015	23
Nougat	7.0 – 7.1.2	Q3-2016	24 – 25
Oreo	8.0 – 8.1	Q3-2017	26 – 27
Pie	9.0	Q3-2018	28
Android 10	10.0	Q3-2019	29
Android 11	11.0	TBD	30

2.1.2 Backward Compatibility

Android releases generally have great backward compatibility; in particular, any app built for recent releases from version 8.0 onward is entirely compatible with the latest API. While for Android versions built for versions between 5.0 and 8.0, inclusion of a simple Dockerfile is enough to install the required packages seamlessly. Moreover, there are tutorials for compiling versions before 5.0 versions of Android [11].

2.1.3 OS Security

Understandably security of any operating system begins with its architecture, which provides the fundamental infrastructure upon which it is built. Android enjoys a multi-layered security architecture, that is briefly touched in this section [12].

Note that while each of these topics are massively extensive in their own right and require comprehensive deliberation to be fully justified, we briefly touch upon each of them in this research to establish a comprehensive understanding of the security foundation of the Android operating system.

Linux Kernel

As previously mentioned, the Android kernel is built upon the Linux kernel [13], the most extensive collaborative software project in existence. Linux kernel is profoundly stable, tested, attacked, and patched repeatedly throughout its lifespan, making it the most trusted OS kernels in the market.

Linux kernel is powered by many security modules [14], which ensure the stability and security of the kernel. The important ones of those is its emphasis on access control and permissions, which is heavily relied on in Android.

Application Sandbox

In addition to Linux kernel security features, Android employs a “sandboxing” mechanism. While there are many different types of sandboxing tools, in general, the sandbox provides an isolated execution environment that disallows access of apps to any part of memory that is not assigned to them by the operating system, effectively restricting apps from modifying or reading data exceeding a specified path [15][16].

In particular, in Android, each instance of the VM functions as a sandbox, effectively executing and restricting its access to another components. This feature provides a powerful and efficient constraint that completely eliminates many security problems without the massive computational overhead. On the other hand, this does not mean that the applications can not communicate with one another at all. The user can allow or deny apps to interact through Inter-Process Communication (IPC), which, can partially nullify the effect of the sandbox. If the user gives improper permissions to IPC, malware may be able to infect the system, and if not, legitimate apps may not be able to function correctly [17].

File System Permissions

By default, no app has permission to execute any process that can harm any code or resources outside of its sandbox. In other words, they cannot access the user’s private data, information stored in other apps, the OS, or connect to the network, microphone, camera, and other resources available to the system. If the app requires access to such resources, it must request the user to accept the relevant permissions in order to gain access. These permissions are stored in `AndroidManifest.xml` file that is shipped in the installer package [8].

System Partition

In addition to sandboxing, to further isolate the components, Android OS employs a partitioning mechanism. By separating the kernel, OS libraries, application runtime, application framework, and apps, the device can execute individual components without invoking other, potentially malicious or corrupted partitions. Additionally, by making the kernel and the OS partition read-only, the integrity of those components can be guaranteed.

Safe Mode

While third-party apps can execute automatically on startup, this can cause obvious security threats. To mitigate this, the OS can be started in a safe mode that prevents any non-essential apps from being automatically launched. Therefore, it inhibits them from harming or disrupting the user experience. This feature also allows the user to detect problems and study individual apps enabling users to troubleshoot the system accurately and prevent apps from recursively disrupt the device.

Verified Boot

Verified Boot is a mechanism which assures all executed codes originate from trusted sources, by establishing a chain of trust ranging from hardware, bootloader, boot partition, and other trusted partitions. When the device boots, the integrity, and authenticity of the next stage are verified before the next execution cycle. The benefit of this system is that it ensures an authentic and secure version of Android is being run, assures that the OS can only be updated to newer versions, and allows the device to inform the user of device integrity [18].

Password Protection

Android allows password protection through encryption via user-supplied passcodes. These passcodes generally protect passwords that are used to encrypt the disk and KeyStore. Additionally, in later iterations of hardware, biometric devices can be used to access the passcodes that enable device usage and disk encryption.

Cryptography

Android devices provide APIs of most commonly used cryptographic primitives to be used by apps and the system, and a keychain mechanism for secure storage of cryptographic keys. These keys can be used for full-disk encryption, full file-system encryption, and file-based encryption and provide cryptographic primitives for apps.

2.2 APK

One of the crown jewels of Android is its incredibly extensive and diverse archive of third-party apps collected in Android PlayStore. These apps developed by the community enable the majority of the capabilities of any mobile device that attracts users to purchase these smart devices.

Android apps come in single packages that can be installed on various devices. These packages are self-containing units with "`.apk`" or "`.xapk`" file format, commonly referred to as APK files. APK is an archive not dissimilar to `.zip` that usually contains at least the following files and folders [19].

- `classes.dex`, classes compiled in `.dex` format readable by the system run-time environment.
- `AndroidManifest.xml`, a file containing metadata about the app, such as name, version, developer name, access rights. An app's ability to access any resources is stored here.
- `META_INF` is a directory that contains:
  - `MANIFEST.MF`, A metadata file containing the names of the files and resources.

  - `CERT.SF` is a list of resources and a SHA-1 message digest of corresponding lines in `MANIFEST.MF`. These are designed to ensure the authenticity of the app.

- `assets`, a directory counting the assets such as icons and graphics.
- `lib`, a directory that contains a set of platform-dependent compiled codes.
- `resources.arsc`, a file containing precompiled resources.
- `res`, a directory containing the uncompiled resources.

2.2.1 Extracting Information From Android Apps

To understand the structure of apps without access to its source code, whether malicious or with intent to study, analyze, and enhance them, methods are required to extract readable codes from the binary files, or application reverse engineering. Reverse engineering in Android is achieved in a way that an app is decompiled, analyzed, and/or modified to a great extent. Figure 2.1 demonstrates an overview of app reverse engineering procedure.

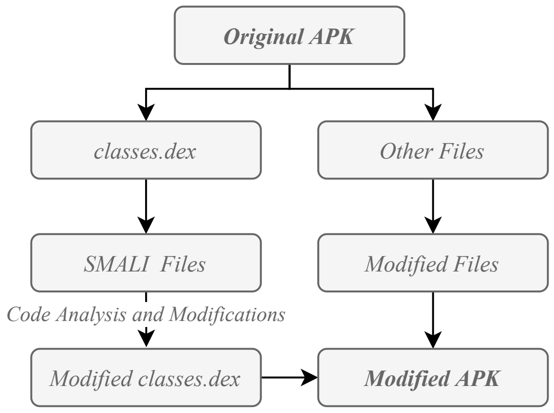


Figure 2.1: Android app reverse engineering overview

While Java classes are packed individually and a `.jar` file can include many classes, APK files have only one `classes.dex` file due to performance and security reasons, as demonstrated in Figure 2.2. This feature is beneficial from a reverse engineering point of view since the target of the analysis is reduced to one file only [20]. Additionally, many apps use third-party libraries; these libraries include individual and separate classes. Upon compilation, such classes are packed in independent `.dex` files and added to the APK file. However, similar to the former, all of the codes imported from each library are packed within one `.dex` file[2].

The first step to reverse engineering apps is decompilation. It is the process of decoding the machine instructions or Common Intermediate Languages (CIL), the instructions that are not understandable by (most) humans, back into source code, effectively reversing the compilation process. However, this process rarely results in the retention actual source code since a great amount of information is lost in the preprocessing,

[2]This procedure begins with an APK file and ends with one. However, for this thesis's purposes and intentions, the process does not need to be completed since a final working app is not required, and only the processes in the left branch are needed.

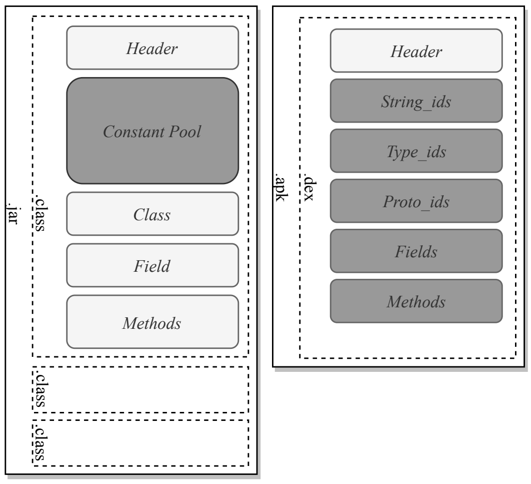


Figure 2.2: Class file vs. DEX file structure

post-processing phases of compilation and obfuscation of information as a protective measure [20].

Android reverse engineering has long been a focus of the scientific community, hackers, pirates, and curious users alike, yet for different reasons. There are many tools and documentation online to effectively decompile and reverse engineer Android apps, each with their advantages and disadvantages. The following list highlights some of the most prominent means of the craft in the market.

- **JADX[3]:** "Command line and GUI tools to produce Java source code from Android `.dex` and APK files".
- **Androguard[4]:** "A powerful python tool to decompile and process Android files with capabilities for bulk analysis."
- **APK Studio[5]:** "A simple GUI software to decompile files, while very useful it is suitable for manual analysis."
- **Bytecode Viewer[6]:**"Six different Java decompilers, two bytecode editors, a Java compiler, plugins, searching, supports loading from classes, jars, Android APKs and more".
- **APKtool[7]:**"A tool for reverse engineering third-party, closed, binary Android apps. It can decode resources to the nearly original form and rebuild them after making some modifications. It also makes working with an app easier because of the project like file structure and automation of some repetitive tasks like building APK, etc."

Interestingly, the scientific community is more interested in methods to increase resiliency and obfuscation of code against reverse engineering and detection of altered code. One of the scarce and valuable resources in the field of reverse engineering is a book

[3]https://github.com/skylot/jadx
[4]https://androguard.readthedocs.io
[5]https://vaibhavpandey.com/apkstudio/
[6]https://bytecodeviewer.com
[7]https://ibotpeaches.github.io/Apktool/

named “Decompiling Android” by Nolan Godfrey [20] which has a detailed explanation of the Android decompilation process. It investigates the matter by looking into the legal, historical, and technical aspects of decompilation.

2.2.2 SMALI

When Android apps are decompiled, the classes are represented in `.smali` files that contain code in SMALI language. SMALI is an assembler for the Dalvik VM and ART bytecode, human-readable representation of `.dex` files. This SMALI code can be analyzed and compiled back to `.dex` files.

Even though SMALI is drastically different from Android’s default top level languages, significant relevance among these languages can be found if close inspection is applied. This relation can be used to detect the app structure and accurately determine the implementations behind them. Chapter 5 exploits this mechanism and studies the methods in which the correct implementation of the apps can be detected.

2.3 Android Security Guidelines

Android enjoys C++ and Java’s solid background as powerful, well-documented, and well-maintained languages, about which many references and documentation discussing their every detail can be found on the Internet. Also, as of 2017 Kotlin programming language[8] has been included as an alternative to Java and is Android’s preferred programming language [21]. This language availability gives Android great flexibility and compatibility allowing developers from many backgrounds to be able to easily start working and producing apps with it.

Android developers, not unlike others, rely heavily on official documents provided by the main developers of the platform, in this case, Android developer guidelines[9]. It includes models to build apps for beginners, sample codes, API references, design guidelines, tutorials, training courses, etc. for various Android enabled devices.

From the security point of view, Android has three major sources of information, (1) Android Developers Blog[10], (2) Android Documentation , and (3) Android Security Bulletin[11]. The security section of Android developers blogs are informal short blogs about changes in Android API and include topics ranging from cryptography to general security implementations, best practices such as how to best implement biometric capabilities for apps. While the documentations follow more formal methods, code snippets, and samples and even though they do not include best practices in particular, includes snippets that have correct implementations of codes. Another difference between the two is that blogs are published once and kept in archive while documentations are regularly updated. The Security Bulletin is a monthly publication that contains details of security vulnerabilities in Android devices which is outside of the scope of this thesis.

As mentioned before, a myriad of tutorials on Android, Java, C++, and Kotlin software development can be found online, but this thesis mainly focuses on Android

[8]https://kotlinlang.org
[9]https://developer.android.com/docs
[10]https://android-developers.googleblog.com
[11]https://source.android.com/security/bulletin

official publications and in very specific cases, we look into some external sources. The reason for this decision is (1) Those are the sole official sources of information that Google approves. (2) There are many external sources which will be out of the boundaries of this inquiry. (3) Many of the sources are incorrect, incompatible, or deprecated, and examining their correctness would not be feasible.

Chapter 3

Cryptographic Preliminaries

This chapter elaborates on the background research in another direction, and focuses on the cryptographic angle of the problem; it aims to explore and justify how cryptography in apps are developed and evaluated.

Section 3.1 focuses on the evaluation criterion that has been chosen to measure the security of apps and how that is justified for app security. Section 3.2 then, establishes rules that ensure the abidance of the apps from said criterion and explains each rule with sufficient detail to stay relevant to the concept.

3.1 IND-CPA Security

To accurately evaluate the cryptographic strength of an app, a unified system of appraisal, that if not followed, the app cannot be considered secure and vise versa is required. Indeed, there cannot be a hard line indicating comprehensive security; for example, an app can follow all protocols of strong cryptography, yet implement a weak password, that arguably, does not concern cryptology, and render the entire effort futile. Despite this, there are cryptographic concepts that distinguish a secure cryptographic scheme from an insecure one. While there exists several of such properties, for this research, we chose to follow IND-CPA security, which we first explain and later justify its conformity for this research.

3.1.1 Definition of IND-CPA

IND-CPA describes situations in which two honest parties share a key k, and the attacker can influence these parties to encrypt messages m_1, m_2, $\cdots$ (using k) and send the resulting ciphertext over a channel where the attacker can observe. In other words, the concept of *Chosen Plaintext Attack* emphasizes scenarios that an adversary has partial to complete control over what is being encrypted [22]. Thus, it is most useful when the attacker can see parts or the entire plaintext as well as the ciphertext.

As IND-CPA is a complex notion comprising various fundamental concepts, few of those must be clarified before it can be formally defined. First, the notion of computational indistinguishability explains the former part of the concept and second the criterion that make a cryptographic scheme secure.

Two family of random variables $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ are computationally indistinguishable if

$$|Pr[D(X_n, 1^n) = 1] - Pr[D(Y_n, 1^n) = 1| \leq \mu(x)$$

while "A function $\mu(x)$:$\mathbb{N} \rightarrow \mathbb{R}$ is called negligible if it approaches zero faster than the reciprocal of any polynomial." [23] or

$$|\mu(x)| < \frac{1}{x^c}$$

Thus, IND-CPA is formally defined as:

> *"A private-key encryption scheme $\prod = (Gen, Enc, Dec)$ has indistinguishable encryptions under a chosen-plaintext attack, or is CPA-secure, if for all* probabilistic polynomial-time *adversaries $\mathcal{A}$ there is a* negligible *function negl such that*
>
> $$Pr[PrivK^{cpa}_{\mathcal{A}\prod}(n) = 1] \leq \frac{1}{2} + negl(N)$$
>
> *Where the probability is taken over the randomness used by $\mathcal{A}$, as well as the randomness used in the experiment."* [22].

To understand the notion, consider an experiment with the following assumptions: (1) Adversary $\mathcal{A}$ has black box access to *encryption oracle* $\mathcal{O}$; (2) $\mathcal{O}$ can encrypt any message with key k inside a blackbox; (3) $\mathcal{A}$ can pass any message m to $\mathcal{O}$ any number of times; (4) $\mathcal{O}$ will always return the ciphertext c of any m as $c = Enc_k(m)$.

For this scheme, the $\mathcal{A}$ is modeled by a *probabilistic polynomial time Turing machine*, meaning that the adversary must complete the experiment and make a prediction in polynomial number of steps. The experiment is as follows:

1. A key k is generated via $Gen(1^n)$;
2. $\mathcal{A}$ generates two messages m and m' with identical length of 1^n uniformly from $b \in \{0, 1\}$ and passes them to $\mathcal{O}$;
3. $\mathcal{O}$ chooses uniform a bit $b \in \{0, 1\}$ and computes $c \leftarrow Enc_k(m_b)$ and returns it to $\mathcal{A}$;
4. $\mathcal{A}$ outputs a bit b';
5. $\mathcal{A}$ outputs 1 if $b' = b$ as success and 0 otherwise.

In other words, if $\mathcal{A}$ is not able to distinguish which one of m or m' has been permutated to c with a probability of more than $\frac{1}{2} + negl$ after that the scheme is IND-CPA secure.

3.1.2 IND-CPA for Android Security

The rationale that makes IND-CPA a sound security baseline for evaluating cryptographic security in mobile apps is that the user is responsible for generating almost all data. Hence, they have access to the plaintext. Any adequately determined adversary can trivially have access to the information before and after encryption. Therefore, if this privilege can reveal any information will compromise the entire security of the app or all other devices that have the same app installed on them. Therefore, the rest of this thesis focuses on possible vulnerabilities that are susceptible to such attacks.

It is noteworthy that IND-CPA (and IND-CCA) are rather theoretical black-box notions that don't have anything to do with specific implementations. If there are any correct implementations of an IND-CPA-secure scheme, then the app should be correct in terms of IND-CPA security. With this in mind, we aim to focus on a series of mistakes in implementation that can lead to vulnerabilities to IND-CPA security and are relevant to Android 10 security status.

3.2 Cryptographic Rules

As mentioned in the Chapter 1, most programmers are not cryptographic aficionados. Therefore, when developing security features in a software, they generally rely on publicly available resources to provide cryptographic primitives they need.

To distinguish the proper use of cryptography, with its misuses, it is paramount to define a set of criteria that explain this distinction clearly. Admittedly, countless bugs and misuses are possible that can lead to IND-CPA vulnerability, and inspecting them all is well beyond the scope of this issue. Therefore, this project uses the set of rules inspired from [6] and adjusts them to the state of Android 10.

Rule 1: Avoid use of ECB (Electronic Code Book) mode of encryption;

Rule 2 Avoid incorrect implementation of IV (Initialization Vector)

- **Rule 2.1:** Avoid use of non-random IV for CBC (Cipher Block Chaining) mode of encryption;
- **Rule 2.2:** Avoid reuse of Key-IV pair in GCM (Galois/Counter Mode) mode of encryption;

Rule 3: Avoid use of constant encryption keys;

Rule 4: Avoid use of constant salts for PBE (Password-Based Encryption);

Rule 5: Avoid use of fewer than $1,000$ iteration for PBE;

Rule 6: Avoid use of static seed in `SecureRandom(.)`;

Rule 7: Avoid use of outdated hash functions.

Following sections will investigate the reasoning behind each rule.

3.2.1 Rule 1: Avoid Use of ECB Mode of Encryption

Standard cryptographic primitives such as AES (Advanced Encryption Standard) operate on various modes; these modes directly affect the indistinguishability of the cryptosystem. Most of these primitives are defined initially with the Electronic Codebook mode of encryption which is the simplest for block ciphers. It processes the plaintexts as individual blocks of a specified size and lacks many of the required security functions. ECB is notoriously insecure against IND-CPA. As demonstrated in Figure 3.1, for any key, each plaintext block permutes to a block of ciphertext. The ECB mode is defined as [24]:

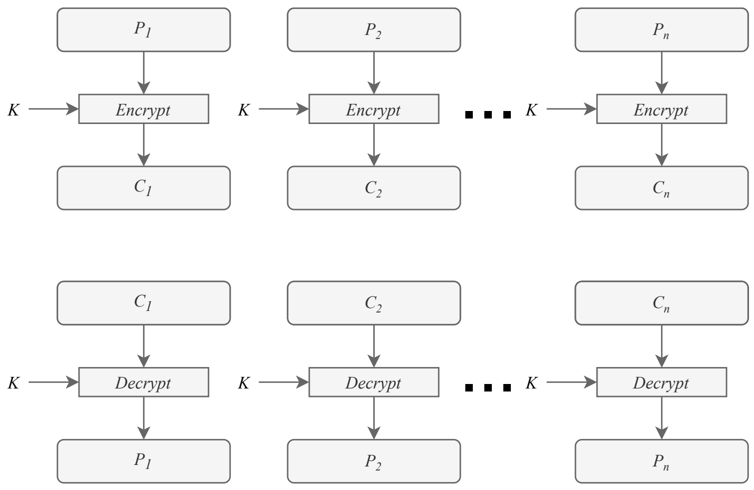


Figure 3.1: Structure of ECB encryption blocks

$$C_i, = ENC_k(P_i) \quad \text{for } i = 1 \ldots n$$
$$P_j, = DEC_K(C_j) \quad \text{for } j = 1 \ldots n$$

As can be seen in Figure 3.1, each block is independently computed with the same key. Thus similar plaintext would produce similar ciphertext, eliminating the probabilistic requirement of an encryption algorithm. Hence, the attacker can trivially extract information about plaintext by observing ciphertext. For this reason, the use of ECB mode of operation is strictly intended for research purposes, and its use in settings with that actual data that needs to be protected is prohibited.

3.2.2 Rule 2: Avoid Incorrect Implementation of IV

In contrast with ECB, there are various secure modes of operation that, if correctly implemented, guarantee the information's secrecy. For example, CBC [25], Cipher Feedback (CFB) [26], Counter Mode (CTR) [27], to name a few, each of which has certain use cases. The common concept in several of these modes is the usage of Initialization Vector (IV) [28], a unique string of characters that provides freshness and randomness to the primitive and can be easily changed without modifying the key.

There are two recommended modes of operation in Android, and since each mode has individual IV requirements, that makes the study of IV specific to each mode, we investigate the correct implementation for these primitives individually.

Rule 2.1: Avoid the use of non-random IV for CBC mode of encryption

In contrast to ECB mode, the CBC mode of operation, if correctly implemented, has robust security guarantees. CBC mode of operation has the advantage of being probabilistic and has avalanche effect [29], meaning an alteration if a single bit will completely alter the consecutive bits. Figure 3.2 demonstrates the structure of CBC encryption and decryption blocks.

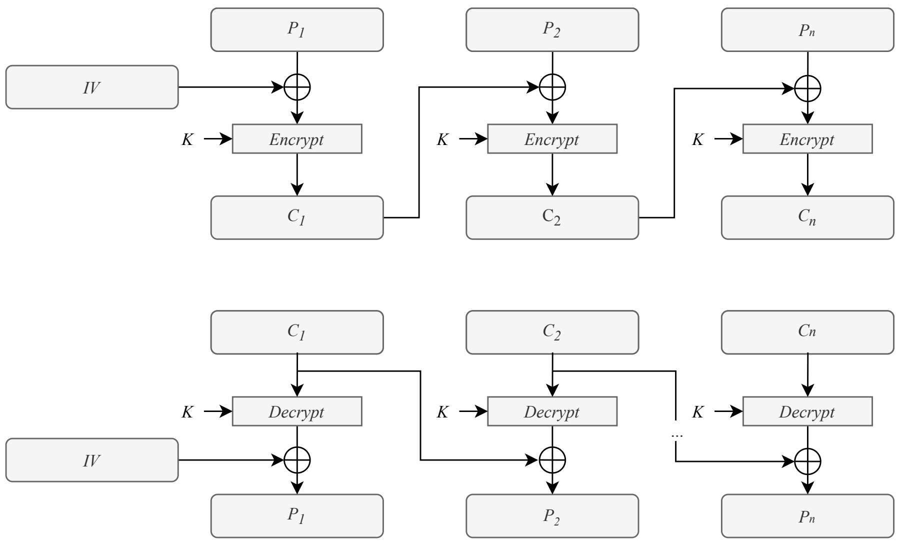


Figure 3.2: Structure of CBC encryption blockscipher

The operation mode contains a mechanism that works in consecutive cycles. During each of which, a block of unencrypted data (p_i) bits is permuted with a key (k) and outputs a block of encrypted data (c_i). The following block then $XOR(\oplus)$ed by the cipher bits of the previous block (p_{i-1}) before its encryption, resulting in a chain of cryptographic blocks that cumulatively affect the consecutive ones [30]. Even though the original patent in 1978 does not include an IV, yet the need for one is apparent, as the encryption of the first block would be deterministic. To remedy this problem, the notion of IV is introduced to add non-determinism, in a way that with every use of the key, a new set of IV bits should be used.

The mathematical representation of CBC scheme is as follows:

$$\begin{aligned} c_i &= Enc_k(p_i \oplus c_{i-1}) \\ p_i &= Dec_k(c_i) \oplus c_{i-1} \end{aligned}$$

where c_0 is always the IV.

If the IV is static, the cipher is susceptible to statistical crypto analysis [31], which can result in key discovery. For instance, using the same IV, key-pair does not result in enough non-determinism, and the beginning of encrypted messages will be ever identical. This property reveals information about the key and can be used in the analysis. Changing the key for every operation is expensive and hence not plausible. Therefore, switching IV for every operation produces enough non-determinism required for resistance to the cipher against various attacks.

Rule 2.2: Avoid reuse of Key-IV pair in GCM mode of encryption

Another one of Android's recommended modes is Counter Mode, which effectively extends the abilities of a block cipher to encompass stream ciphers without compromises inherent to classic stream ciphers. One of the most prominent modes of encryption with this ability is Galois/Counter Mode (GCM), which is the cutting edge encryption mechanism of secure high-speed communication with inexpensive hardware [32] and offers built-in authentication, requires no padding, bares identical ciphertext to plaintext length, has low memory requirement, and holds arbitrary message length(up to $2^{39} - 256$) bits [33].

Figure 3.3 demonstrates the structure of CBC encryption. Note that an IV is initiated, enumerated as a counter, and then encrypted with the key, and while unlike CBC, each cipher block is independent of the former. Nevertheless, due to the counter, each block is effectively permuted with a different key-counter combination. Another benefit of GCM is that given key and the counter the entire key-stream can be generated in advance while computing the cipher/plaintext is a matter of an $XOR(\oplus)$ operation, this enables parallelism and high-speed computation [33]. In addition, GCM mode offers built-in authentication, requires no padding, identical ciphertext to plaintext length, low memory requirement, arbitrary message length(up to $2^{39} - 256$) bits [33].

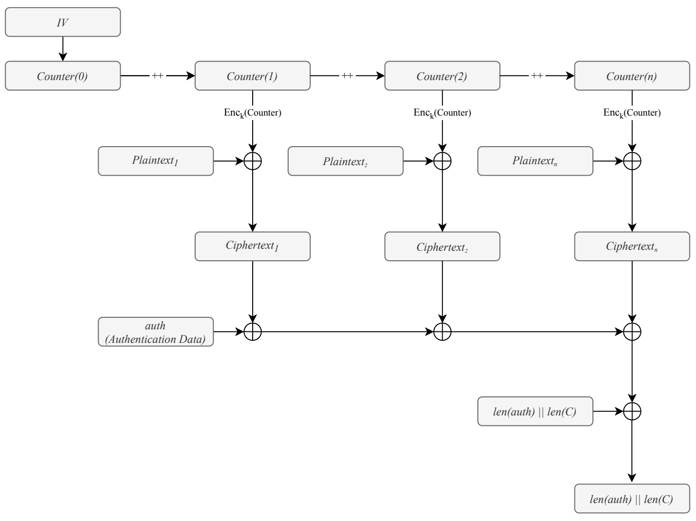


Figure 3.3: Structure of GCM encryption blockscipher

To ensure secrecy of GCM, some organizations like National Institute of Standards and Technology (NIST) put rigorous requirements on the proper use of GCM mode in contrast with others [34] since it is particularly vulnerable to Key-IV pair reuse. As the plaintext is XORed with the encrypted IV (N.B. IV is not secret), to produce ciphertext, reuse of the combination can be devastating to the scheme's security.

3.2.3 Rule 3: Avoid the Use of Constant Encryption Key

Arguably, secret keys are the most fundamental elements in symmetric cryptography. They have existed, in some form, from cryptography's inception, and its secrecy is the underlying assumption in security. As such is Shannon's maxim which is the reformulation of Kerckhoffs's principle that indicates: "one ought to design systems under the assumption that the enemy will immediately gain full familiarity with them" [35] meaning that the security of a ciphertext should not rely on the assumption of secrecy of algorithm and the key remains the sole secret part of a cryptosystem.

There are several methods to communicate and create secret keys between parties, one of the most ubiquitous of which is *Diffie–Hellman key exchange algorithm* [36] that effectively and discretely creates secret keys between two parties.

Using a constant secret key while solving key management problems creates widespread vulnerabilities in apps. Since generally as described in the incoming Chapter, decompiling and finding own secret keys in an app is quite trivial, it enables an adversary to effectively understand the key that is used the same app in all other devices.

3.2.4 Rule 4: Avoid Use of Constant Salts for PBE

To achieve non-determinism in cryptography, often an external source of randomness is required, similar to the concept of IV, discussed in Section 3.2.2, many cryptographic schemes are designed in a way that randomness is separate from the actual algorithm to ensure freshness and tweak-ability. Similarly, this principle holds for password-based encryption. Especially since the passwords are often chosen from groups with relatively small entropy, there is a need for extraordinary attention when it comes to protecting password databases against brute force attacks.

As described in [37], a *salt* is a random value (Morris et al. recommend using the system's internal clock for this operation) appended to the password and saved with the file which does not increase the computational task required to find any specific password. Still, it significantly increases the workload demanded by testing large databases.

The main advantage of using salt is that it does not require to be kept secret. This is because even if the adversary has access to the database, salt, and a list of potential passwords (such as hash-lists [38] and Rainbow tables [39]) creating a list of possible keys will have to be done individually for each password which is drastically more intensive for the adversary [40].

Figure 3.4 further elaborates on the concept, as it can be seen if there are similar passwords in the database, if no salt is used, they yield identical output. Hence, if the adversary has access to a pre-computed password database or computes one of them, all related entries are revealed. However, if a salt is used, the outputs are different. Consequently the adversary is required to completely recompute the password[1].

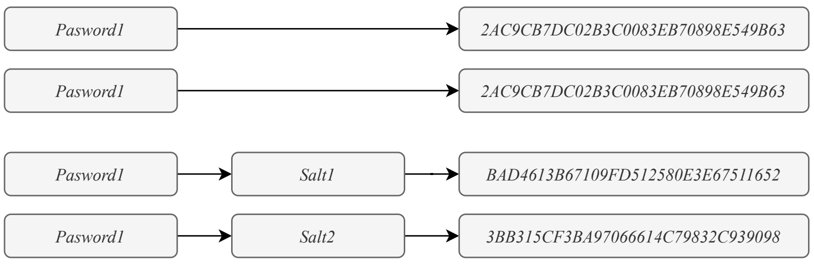

Pasword1
2AC9CB7DC02B3C0083EB70898E549B63
Pasword1
2AC9CB7DC02B3C0083EB70898E549B63
Pasword1
Salt1
BAD4613B67109FD512580E3E67511652
Pasword1
Salt2
3BB315CF3BA97066614C79832C939098

Figure 3.4: Effect of salt in PBE

The use of constant salt partially nullifies these advantages. Even though it is still likely resistant to pre-computed hash tables, using a constant salt would result in identical passwords being identical after the encryption, ending in a reduction in the time required

[1]This figure is for demonstration purposes only. This paper does not promote the use of a weak password, salt, or weak hash algorithms for password-base encryption.

for computing passwords.

3.2.5 Rule 5: Avoid Use of Fewer Than $1,000$ Iteration for PBE

Computation speed is one of the considerations in cryptography. Cryptographers, developers, and hardware manufacturers are always trying to find ways to accelerate encryption and decryption of data. However, this does not hold for PBE since fast computation allows attackers to precompute massive databases quickly. One of the measures to circumvent this is to repeat the encryption maximum tolerable number of rounds.

The number of iterations is selected because it must be the maximum number of iterations that the system can handle in a way that reduces the computation time for the attacker but is not noticeable by legitimate users. [40] indicate the number of iterations to be 1000[2].

3.2.6 Rule 6: Avoid Use of Static Seed in SecureRandom(.)

A good Random Number Generator (RNG) sits at the heart of good cryptography, as almost all cryptographic protocols require produce and useing secret, non-predictable values. For example, RNGs are used in symmetric, asymmetric, and hybrid key generation, IV generation, Key Derivation Functions (KDF), used to create challenges, nonces, salts to name a few.

One of the critical conditions that a suitable random number generator should abide is its unpredictability. Meaning no adversary, even those with intimacy with RNG's internal design, and vast computational capabilities must be able to make a useful prediction about the RNG function's output. In other words, a proper RNG output should have an apparent entropy as close to the bit length as possible [41].

Shannon [42] defines entropy H of a bite string as:

$$H = -K \sum_{i=1}^{n} p_i \log p_i$$

where K is an optional constant (e.g., $1/\log(2)$), p_i is probability of occurrence of i out of n possible states.

There are two types of RNGs. True Random Number Generators (TRNG) use a non-deterministic source of entropy; measurements often obtain that from unpredictable natural phenomena such as nuclear decay [43], mouse movement, and chaotic hash functions [44], [45], etc. Pseudorandom Number Generators (PRNG) uses deterministic computational chains to generate a series of outputs from a seed that looks random. Yet, the entropy of the output cannot surpass the entropy of the seed. Nonetheless, a good PRNG can be computationally impossible to distinguish from a TRNG [41]. A PRNG must use a TRNG seed to initiate its sequence to be considered secure since it is infeasible to create true randomness with a deterministic system [41].

[2]Note that the Android Keystore system is in charge of key storage and protection in Android devices. However, a substantial volume of the documentation is attributed to PBE, it is still supported by Android, and is an important cryptographic primitive for IND-CPA. Therefore, the inclusion of PBE seems to be justified.

3.2.7 Rule 7: Avoid Use of Insecure Hash Algorithms

Cryptographic hash functions are a series of one-way algorithms that take in data of arbitrary size and output blocks of seemingly random, fixed size, also known as a message digest with the condition that identical input always returns the same output. Hash functions have a wide array of uses in cryptography such as Message Authentication Codes (MAC), Digital Signatures, commonly used in cryptocurrencies, and many more areas.

Figure 3.5 demonstrates an example of the input-output relation of a hash algorithm. Note that a single input always yields the same output while any changes in the input alters the output completely.

Hash algorithms must have two characteristics: (1) they must be one way, meaning that given hash function $H(x)$, it must be impossible to compute x, and (2) it must be impossible to find a message x' that $x' \neq x$ such that $H(x) = H(x')$ also called collision resistance [46].

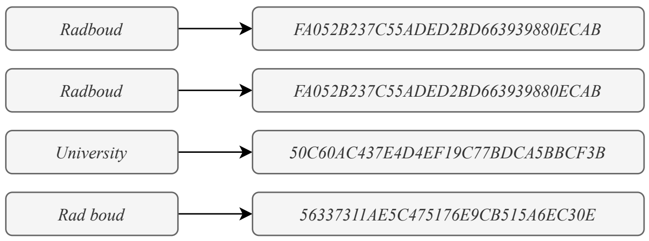

Radboud
FA052B237C55ADED2BD663939880ECAB
Radboud
FA052B237C55ADED2BD663939880ECAB
University
50C60AC437E4D4EF19C77BDCA5BBCF3B
Rad boud
56337311AE5C475176E9CB515A6EC30E

Figure 3.5: Example of execution a hash algorithm

There exist several standardized hash functions, even though all of which are practically impossible to reverse, due to information loss, some are no longer collision resistant as there have been vulnerabilities discovered that could cause an adversary to find collisions and effectively nullify the second requirement of hash function security [47].

Chapter 4

Related Work

Due to the ubiquity of Android devices, accompanied by the facts that many of its services are open-source, and it is built upon the foundation of Java as a very well established language it enjoys ample accessibility. Naturally, following this trend many scientists have been studying every aspect of Android's working mechanisms. This chapter reviews some of the most notable research articles relevant to this topic.

With Android's massive user base, it is natural that it has piqued the interest of different groups in academia and otherwise, and its cryptography is no exception. The cryptographic security of Android's apps has been well explored and studied throughout its existence, and numerous papers have been published in this regard.

One of the most prominent studies in the field is the groundbreaking paper by Egele et al. [6] published in 2013. The research defines six rules regarding the IND-CPA security of Android apps with security standards and guidelines relevant at the time of publication. The study investigates whether cryptographic APIs are correctly implemented and adequately satisfy the required cryptographic concepts.

To justify the research with sufficient empirical evidence, the research employs an analysis technique to automatically check apps with the aid of a customized program called CryptoLint. CryptoLint uses program slicing [48] and construction of Control Flow Graphs (CFG) [49] to detect misuses in Android apps. In the end, the research concludes that in total, 88% of the apps in the market make at least one misuse that causes vulnerability to IND-CPA security.

In the following year, Shuai et al. in [50] continue the work by expanding the number of rules to 18, investigating misuses in asymmetric and symmetric encryption, hash algorithms and key management while drastically reducing the number of apps analyzed to 45. The study fails to examine all the specified rules due to the complexity of the matter. It merely examines symmetric and asymmetric encryption and hash misuse. To analyze apps, the authors created a program called CMA (Cryptographic Misuse Analyzer) which first runs static analysis on each app to determine if cryptographic APIs are used. Then constructs the CFG and Call Graph (CG) [51] of the app and determines which branch the cryptographic APIs are located in and if they are implemented correctly at the end of the research, it reasons that only 5 out of the said 45 selected apps where cryptographically secure. Arguably, the study suffers from a very small sample space of the apps examined. The research article, however, employs unusual methods for its analysis.

In 2015, Chatzikonstantinou et al. in [52] follow a similar approach. By incorporating 24 rules and examining 49 apps and using a mixture of static and dynamic methods to detect the cryptographic misuses, they conclude that 87.8% of the said apps were cryptographically insecure and the rest employed no cryptography at all. The paper does not mention the number or percentage of correctly implemented apps if there was any.

Ma et al. in [53] follow the steps laid by Egele et al., by slightly updating the rules to include the use of outdated hash algorithms. The focus of the work is the development of a program to s, repair and repack vulnerable apps by using a very efficient mechanism to detect misuses by identifying the signs of mistakes in the source code. They applied the software to 8640 apps and successfully repaired 94.5% of the faulty apps in their dataset. Ma et al.'s research inspires the detection mechanisms used in this study.

Muslukhov et al. [54] conduct a study with a focus on source attribution analysis cryptographic misuse in Android apps. The study aims to clarify if the cryptographic mistakes mentioned in the previous studies are originated from code written by application or third-party library developers. They developed a static program analyzer that supports source attribution, analyzed 132,000 apps, and concluded that third-party libraries are the primary source of cryptographic misuse.

Chapter 5

Investigation into Android Documentation

To answer the first research question, *"How comprehensive, usable, and reliable is the Android documentation regarding implementation of cryptographic libraries in Android apps?"*, it is necessary to delve deep into Android documentation concerning cryptography and scrutinize all documentation that touches each of the rules. For do so, we take each of the rules in 3.2, scan the entire documentation for any related article and thoroughly analyze the document.

5.1 Improper Modes of Operation (Rule 1)

Android documentation and guidelines regarding symmetric cryptography [55] recommend using *AES* in either *CBC* or *GCM* mode with *256-bit* keys and *no padding.* The documentation supports this with a few code snippets found in [56]. Sample codes are adequately descriptive, easy to use, and are written in both Java and Kotlin. However, avoidance of ECB mode of operation has never been mentioned in Android developer sources, nor has it offered any remediations for the programs that fail to indicate a mode of operation, or specify ECB in the `getInstance(.)` method of `Cipher` class which leads to ambiguity.

Interestingly, there is not much information about insecure implementations of symmetric cryptography of Android on other online sources either. While many tutorials explain the implementation of various modes of operation none of them emphasizes the insecurity of this mode. Coupled with the fact that it is the simplest mode of operation (since it does not require an IV) and on many occasions the first mode of operation that is explained as introduction to the encryption mechanism, this can lead to many misuses.

The documentations fail to clarify the methods' default behavior as well. Thus, if a developer does not indicate the exact desired mode of operation, the mode of operation it employs or the exception it causes is generally unclear. Even though this is consistent throughout the documentation, it can always lead to ambiguity. Many Java Cryptographic providers (such as Bouncy Castle which was the default provider until Android 9 and was deprecated in 2018) have ECB as the default mode of operation [57].

5.2 Improper IV Usage (Rule 2)

As mentioned previously, IV plays a significant role in the security of many modes of operation. As a small mistake in its implementation can cause vulnerability in the scheme's IND-CPA security, coupled with the general ambiguity and complexity of the concept of IV, it is paramount that any documentation explains this idea articulately.

There are guidelines on cryptography that briefly touch on the concept if IV. Such as in [56] while demonstrating message encryption code sample, in Java and Kotlin there is a line that illustrates how the IV should be implemented. Curiously, this sample does not attempt to express how to decrypt messages that have been encrypted via this method. This shortcoming may cause incompatibility and failure resulting in the developers relying entirely on insecure schemes found online thus, nullifying the entire effort. Additionally, in the section "PBE ciphers without an IV", in the same entry, it mentions that it can be obtained either from an explicitly-passed IV or if adequately constructed, from the key in PBE ciphers that require one. However, when passing a PBE key that does not include an IV and does not explicitly give one, the PBE ciphers on Android currently assume an IV of *zeros*. Even though this is the default behavior for many JCPs, it is nonetheless an enormous vulnerability in CBC mode. Since the IV is static a wide array of different users would suffer from identical IV in the encryption mechanisms of their apps that can cause mass exploitation.

The code sample provided in the same document does not follow this rule and implements the algorithm with default IV, meaning that any application that uses this code is vulnerable.

Additionally, even though GCM is one of Android's recommended modes of operation, with stringent IV requirements and dangerous pitfalls, the documentation completely ignores it. It entirely fails to address the best practices of this mode, since it is convenient for large amounts of data and streaming purposes, its compromise can have catastrophic outcomes.

5.3 Constant Secret Key (Rule 3)

In Android OS, keys are managed via the Android Keystore System [58]; it allows developers to store keys in a container that cannot leave the device and can be used for various cryptographic operations [59]. The features of Android Keystore System are (1) Extraction prevention, (2) Hardware security module, and (3) Key use authorizations.

Android documentation also provides code samples, explanations, and best practices on how to generate various fundamental mechanisms in both Kotlin and Java [60]. However, it does not emphasize or prohibit developers on key reuse or using constant keys. But, developers could use code samples that are provided (and recommended) in the documentation.

5.4 Constant Salt in PBE (Rule 4)

Android documentation explains that the salt should be a random string [61] generated using `SecureRandom` and stored in internal storage alongside any encrypted data to prevent attacks using precomputed hash and Rainbow tables. Coupled with the information explained in section 5.6, it is safe to say that the concept of using a random salt is appropriately covered by Android developers guide.

Furthermore, in the entry [62], a code sample in Java for implementation of PBE, with proper salt is provided. Nevertheless, curiously it has not been treated with much care as it does not explicitly mention that it is indeed a PBE and merely mentioned that it is how encryption should look like. Also, the code sample hardcodes the password (which, as discussed in Section 3.2.3, is in itself insecure). There is a link in the same paragraph that redirects to an external tutorial with a complete explanation of a sample PBE encryption.

It is noteworthy that the requirement for salt is weaker than what is mentioned in the documentation [63], and it is enough if the salt is random (as opposed to secure random). However, for the sake of consistency, we choose to follow Android documentation for experimentation.

The documentations provide code samples in [62] that is a complete implementation of the algorithm. However, the documentation uses an outdated encryption primitive (`PBKDF2WithHmacSHA1`) and should be changed to “`PBEWITHSHA256AND256BITAES`” as mentioned in [56]. Additionally, the original code includes a syntax error. Since these are the sole source of information, the inconsistencies and syntax errors can confuse novice programmers.

5.5 Low Iteration PBE (Rule 5)

Since this particular misuse is employed in parallel with the one in Section 3.2.4, the complete documentation of the two is entirely similar. The only related highlight in the entry [61] is that it includes a Java code sample that explains “Larger values increase computation time. You should select a value that causes the computation to take > 100ms”.

5.6 Bad Random Seed (Rule 6)

Following the points highlighted in this article [64], Klyubin reports Java Cryptographic Architecture (JCA) is not secure for key generation, signing, or random number generation in Android devices due to “improper initialization of the underlying PRNG” [65]. To remedy this problem, the entry recommends developers to use PRNG with entropy from `/dev/urandom` or `/dev/random` which by itself is insecure as discussed in Section 3.2.6 since the source of randomness must be a TRNG seed. However, reportedly as of June 2016, despite the guidelines mentioned in existing documentation [66] many apps still used the SHA1PRNG algorithm from the “Crypto” provider.

The usage of the SHA1PRNG algorithm is not recommended since it is not cryptographically safe as “the random sequence, considered in binary form, is biased towards returning 0s, and that the bias worsens depending on the seed” [67]. The Crypto JCP only offers `SecureRandom(.)` functionality with SHA1PRNG, hence, it is not considered a safe provider.

Similarly, a common but incorrect method to acquire a seed was (by the time of writing the blog) only using a password. This method has two disadvantages. First, password was typically a constant for a reasonable amount of time. Second, the implementation of SHA1PRNG in Crypto provider had a bug that if the setSeed(.) method were called before obtaining output, the result would be deterministic [62].

To help developers create safer apps and recover from previous incorrect implementation, the following is provided by the entry [62]:

- The correct use of the key derivation mechanism, and how to decrypt data that was encrypted with an insecure key with an example that can be found in the documents[1].
- Correct methods for deriving keys, namely:
  - Reading AES keys from a disk: the user can store the keys on the device since it is protected by the disk encryption mechanism and system permissions. In contrast, the external storage is not protected by these measures [61].
  - Generating the keys by: `<SecretKey key = new SecretKeySpec(keyBytes, "AES")>`
- A helper class `called <InsecureSHA1PRNGKeyDerivator>` to derive keys from insecure implementations that drove the key from a password every time the code samples can be found in the documents. The developer can then re-encrypt data with correct keys.

Lastly, the entry indicates that the Android returns an instance of `OpenSSLRandom`, a reliable source of randomness derives from `OpenSSL` if an example of SHA1PRNG is requested without a specification of a provider.

It is noteworthy to mention that the default behavior of this SecureRandom(.) is that it automatically generates a secure seed from system entropy. Therefore, even though this particular rule is more natural to implement correctly and its misuse requires extra, unnecessary steps, we chose to include it as it can establish whether primitives with secure default behavior can lead to fewer misuses.

5.7 Outdated Hash Algorithms (Rule 7)

While Android cryptographic libraries support several Cryptographic Hash Functions (CHF), some of which are no longer considered secure [68], the use of deprecated hash functions is by no means prevented in Android documentation. Table 5.1 [69] demonstrates all supported hash algorithms and their current state of security against collision attacks. As can be seen in Table 5.1, some hash algorithms are not secure anymore and are still supported by Android. Note that the table only considers collision resistance[2].

The sample code in [69] includes a code sample that implements a hash function. It allows the developers to replace the hash algorithm with their desired option from the list. Considering that there are two insecure options, this could yield too many misuses of this kind. On the contrary, [56] introduces a code snippet that correctly implements a secure hash function using `SHA-256` algorithm.

[1] https://android.googlesource.com/platform/development/+/master/samples/BrokenKeyDerivation

[2] A collision attack happens when an attacker finds two different messages that can be hashed to the same digest.

Table 5.1: List of supported hash algorithms and their relative security

Algorithm	**Supported**	**Secure**
MD5	✓	✗
SHA-0	✗	✗
SHA-1	✓	✗
SHA-224	✓	✓
SHA-256	✓	✓
SHA-384	✗	✓
SHA-512	✗	✓
SHA3-224	✗	✓
SHA3-256	✗	✓
SHA3-384	✓	✓
SHA3-512	✓	✓

Chapter 6

Experimentation

To determine how accurately the app developer community follows the documentation and to answer the second research question *"How effective have the changes in the documentation been on cryptographic correctness of apps in the Android PlayStore?"*, a series of experiments are required, i.e., an adequate number of apps should be randomly selected, decompiled, and studied for traces of code that represent the Rules. Subsequently, the result of the experiments must be summarized to determine an accurate estimation of the development culture around the Android app ecosystem.

It is noteworthy that, even though Kotlin is the preferred language of Android development as of 2020, we chose to follow Java for these experimentations for three main reasons (1) it is a better-established language throughout the documentation and (2) upon initial inspection it is evident that there are more code samples for Java in Android documentation (3) when decompiled to SMALI, both languages yield similar code with minimal changes that can be easily circumvented.

In this chapter, we design an experiment that will analyze a dataset of apps from the Android Playstore. To do so, first in Section 6.1, we describe the general structure of the experiment and the reasoning behind it, and in all the subsequent sections, we explain every step in detail.

6.1 Experiment Structure

As presented in Figure 6.1, the experiments have been designed to have a modular and sequential structure, meaning that each step begins after the previous one is completed. Even though this approach increases the duration of the experiment to a certain extent and limits the program's multiprocessing abilities, it is beneficial to each module so it can be fine-tuned if problems arise.

6.2 Dataset Acquisition

Finding a proper source of Android apps would prove to be a challenge. Mainly because none of the official Android repositories offer bulk access to apps for research purposes. However, Allix et al. [70] have gathered an expansive archive of millions of Android apps

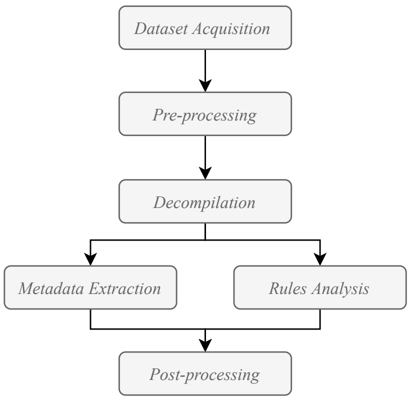


Figure 6.1: Experimentation steps

from various sources including Google Play[1], Anzhi[2], AppChina[3], etc. and grant access to the research community.

An appropriate selection of dataset would prove crucial to assess the app ecosystem accurately. The primary criterion for this selection is: (1) apps should be from authorized sources, and (2) apps must be legitimate and real. We chose apps from Google Playstore and those which are not known to be malware for our investigation. Note that there may be well-hidden malware in the selection, but the target list is set to exclude them.

The way that Androzoo database can be accessed is through an API key that was granted after a request. Afterward a designated list of all apps in a `.csv` file was downloaded, and the following call to Androzoo was used for this process:

```
    az -n [number of apps] -d [publication date]: -s :[maximum size] -t [
    number of threads (4)] -m  [source]
```

Listing 6.1: Call to Androzoo

where maximum size was chosen to be 10MB, and apps were only selected from Google Play Store.

6.3 Pre-Processing

Androzoo's download script comes with a few caveats: first, it cannot download apps within a specific period. It can only download from the designated date onwards, while this is not a problem for apps developed since 2020; for earlier years, the number of apps downloaded was uneven. To solve this problem, the pre-processor was designed so that if a certain period had more than the required amount, it was dropped and moved to the next app. Second, there could be duplicates in the downloaded dataset since the index of the app in the Androzoo database was randomly generated. In some years, especially in 2020, the number of apps was smaller, which could yield many duplicates. Therefore, the pre-processor had to remove duplicates as well.

[1]http://play.google.com
[2]http://www.anzhi.com
[3]http://www.appchina.com

This process was generally wasteful since many apps were downloaded and discarded, as there was no convenient way to determine what app was being downloaded next. However, after the pre-processing phase, a precise number of apps from each year could be assembled which was adequate for the project.

6.4 Decompilation

After the dataset acquisition, the apps must be decompiled; this was done with a Python script that loops through the `.apk` files and runs the decompiler command. While this is a simple process to program, it is indeed the most time consuming; therefore, it could greatly benefit from multiprocessing.

Various decompiler tools provide powerful solutions to decompiling APK files. After much deliberation, we chose to use APKtool for its simplicity, flexibility, and good quality of the source file it generates. Very simply, APKtool takes an APK file and decompiles it to a bundle with classes and resources encapsulated in the APK additionally, as it can be called on an app. At the same time, the entire decompilation process is abstracted away.

The following bash command was called on each file to proceed with the decompilation:

```
apktool d [apk_file]
```

Listing 6.2: Decompilation command with APKtool

6.5 Rules Analysis

The most important part of the experiment arguably, is the detection of the aforementioned misuses. In this phase, the rules must be codified so as to be applied to the "`.smali`" files of the apps and detect if the rules are misused.

For this analysis, we used a python script, with techniques such as program slicing [48], and regular expressions (RegEx) [71]. First, it finds instances of each rule with the aid of an "endpoint signature"; this is a string that, if present, the program continues processing the file or ignoring it if otherwise. For example, if the code contains string "MessageDigest", then the app uses a hash function. Afterward, the program checks for "Slice pattern", which finds the point program which is to be sliced; this point is usually a method that its parameters distinguish how it is used. For example, in "`(v1, v2 ), Ljava/security/MessageDigest`" the first parameter is the text to be hashed, and the second is the hash algorithm. Lastly, the program searches the slice for "Misuse Pattern" which means that it seeks where that variable is assigned, and if that assignment includes a misuse, it marks the app faulty.

This explanation is indeed a general description of the entire detection algorithm, yet each rule has its intricate details that require explicit attention. The following subsections explain those with greater attention without repeating itself.

It is noteworthy that, even though Kotlin is the preferred language Android development as of 2020, we chose to follow Java for these experimentations for three main reasons (1) it is a better-established language throughout the documentation, (2) upon initial inspection it is evident that there are more code samples for Java in Android documentation, (3) when decompiled to SMALI, both languages yield similar code with minimal changes that can be easily circumvented.

6.5.1 Rule 1

As mentioned before, Android documentations specify a code snippet for correct implementation of an AES in [56]:

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING"); //The encryption algorithm, mode of operation and padding scheme.
```

Listing 6.3: Correct implementation of AES in Java

Listing 6.3 is the line of code that distinguishes the encryption mechanism recommended by Google.

Upon close inspection, in the file `MainActivity.smali` the following SMALI code can be found:

```
 const-string v4, "AES/CBC/PKCS5PADDING" //Equivalent to line 5 of the Java program above
invoke-static {v4}, Ljavax/crypto/Cipher;->getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;
```

Listing 6.4: Correct mode of encryption SMALI

The `invoke-static{}` of `Cipher;->getInstance(.)` method takes a `String` parameter `v4` which is set on the previous line indicating the mode of operation.

To understand the discrepancies between the correct and incorrect implementation, the following changes have been incorporated to Listing 6.3: ,

```
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING"); //Note that the mode of operation has been changed to CBC
```

Listing 6.5: Incorrect implementation of AES encryption in Java

which yields the following change in the SMALI code:

```
 const-string v4, "AES/ECB/PKCS5PADDING"
invoke-static {v4}, Ljavax/crypto/Cipher;->getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;
```

Listing 6.6: Incorrect mode of operation in SMALI

Comparison between the Listings 6.4 and 6.6 shows the contrast between the correct and incorrect modes of operation in SMALI. To accurately detect this misuse, the following arguments were used to the detection method

Table 6.1: ECB misuse detection parameters

Endpoint Signature	`Cipher;->init`
Slice Pattern	`{(v[1-9]+)},\s\w*/crypto/Cipher;->getInstance`
Misuse Pattern	Algorithm placeholder + `"[A-Z]+/([A-Z]{3})/[A-Z0-9]+"`
Misuse Indicator	`"ECB"`

As previously mentioned, many of the detection algorithms' details are approximately similar therefore for the sake of conciseness, from this section on, the information about the code will only be elaborated if the program does use a significantly distinctive mechanism.

6.5.2 Rule 2

To detect if the IV is correctly implemented, the `Cipher;->init` method should be examined regardless of the mode of operation. The method signature has required values for the algorithm (e.g. "`AES/CBC/PKCS5PADDING`"), operation constant (e.g. `ENCRYPT_MODE` or `DECRYPT_MODE`) and the key. Listing 6.7 shows, the relevant SMALI line.

```
invoke-virtual {v3, v5, v0, v4}, Ljavax/crypto/Cipher;->init(...
```

Listing 6.7: Smali Code for cipher algorithm

The procedure must only be followed if the algorithm is in encrypt mode since it is natural for decrypt mode to receive the IV (or key) from an external source. Additionally, while the parameter for IV is optional, as mentioned before, in Android, if no IV is correctly passed, the IV of zero will be taken into consideration. Hence, a part of the detection of this misuse is that the number of arguments passed to `Cipher;->init` method should be more than three.

Table 6.2: IV misuse detection arguments

Endpoint Signature	`Cipher;->init`
Slice Pattern	`{([A-Za-z0-9 ,]*)\},\s\w*/crypto/Cipher;->init`
Misuse Pattern (Enc)	Encryption constant + `r',\s0x(\d)`
Misuse Pattern (IV)	IV parameter +`},\s\w*/security/SecureRandom;-><init>`

6.5.3 Rule 3

Key generation in Android is done via `KeyGenerator` class, as seen in Listing 6.3. First, an instance of the key generator is created through `keyGenerator.getInstance(.)` with the encryption algorithm as a parameter, second the length of the key is established via the `init(.)` function, and then the key is generated with `generateKey(.)` method.

The relevant part of the SMALI code for key generation in SMALI is as follows:

```
  const-string v4, "keygen.generateKey()"

  invoke-static {v3, v4}, Lkotlin/jvm/internal/Intrinsics;->
  checkExpressionValueIsNotNull(Ljava/lang/Object;Ljava/lang/String;)V
```

Listing 6.8: Smali code corresponding to key generation

Table 6.3: Constant key misuse detection parameters

Endpoint Signature	`Cipher;->init`
Slice Pattern	`{([A-Za-z0-9 ,]*)\},\s\w*/crypto/Cipher;->init`
Misuse Pattern	Key variable hook + `},\s\w+/crypto/KeyGenerator;->init`

Detection of this misuse involves finding all the instances of `Cipher;->init`, examining its parameters, and ensure that the third parameter (see Section 5.1) does initialize with `KeyGenerator` class and not from any other source. Note that there are indeed other methods for key generation (e.g., Diffie-Hellman key exchange and key derivation functions such as HKDF). Still, there is no trace of such algorithms in Android documentation; while it is not ideal, the detection method focuses on what is provided by Android. Another important note is that Android allows keys to be extracted and used from the Keystore system. However, security assessment of the Keystore is out of project scope, and calls to the Keystore system are marked as such.

6.5.4 Rule 4

The relevant section in `MainActivity.smali` file is as:

```
        invoke-direct {v6, v7, v5, v1, v3}, Ljavax/crypto/spec/PBEKeySpec
;-><init>([C[BII)V
```

Listing 6.9: Corresponding SMALI code to salt implementation

where the third parameter is the seed,

```
.local v4, "salt":[B
new-instance v5, Ljavax/crypto/spec/PBEKeySpec;
```

Listing 6.10: Salt assignment in SMALI

Thus, to detect the misuse, the 3rd parameter must be identified and tracked. If the argument is instantiated with an instance of `SecureRandom` the program is correctly implemented.

Table 6.4 contains the parameters passed to the detection method.

Table 6.4: Salt misuse detection parameters

Endpoint Signature	`PBEKeySpec`
Slice Pattern	`{([a-z0-9\s,]*)},\s\w+/crypto/spec/PBEKeySpec`
Misuse Pattern	salt variable + `},\s\w+/security/SecureRandom`

6.5.5 Rule 5

Similar to detection of salt, the initialization of the 4th parameter must be initialized with a value of more than 1,000. If the hexadecimal representation of the assigned value is less than 0×3E8, the app is incorrectly implemented.

```
const/16 v1, 0x64
```

Listing 6.11: Low iteration count in SMALI

Table 6.5: Iteration count misuse detection parameters

Endpoint Signature	`PBEKeySpec`
Slice Pattern	`{([a-z0-9\s,]*)},\s\w+/crypto/spec/PBEKeySpec`
Misuse Pattern	itter var hook + `,\s(0x[a-z0-9]+)`
Misuse Indicator	itter $\leq$ 1000

6.5.6 Rule 6

Detection of this misuse is arguably the simplest of all. As mentioned before the default behavior of `SecureRandom` is implemented with the secure seed and the programmer should directly use `SetSeed(.)`.

To detect this misuse, the entire code has to be analyzed, if there is an instance of SecureRandom and `SetSeed(.)` is called, it is evident that the app has this misuse in its implementation.

Table 6.6 demonstrates the arguments passed to this method.

Table 6.6: Seed misuse detection parameters

Endpoint Signature	`SecureRandom`
Misuse Pattern	`setSeed`

6.5.7 Rule 7

Detection of misuse of the 7th rule is very simple and similar to the algorithm used in Section 5.1. the only difference is the arguments that have been passed to the algorithm, which are presented in Table 6.7 :

Table 6.7: Hash misuse detection arguments

Endpoint Signature	`MessageDigest`
Slice Pattern	`{([a-z][0-9]+)},\s\w+/security/MessageDigest`
Misuse Pattern	hash algorithm placeholder`,\s"([A-Z0-9-]*)`
Misuse Indicator	`SHA-1`
	`MD5`

Additionally, to ease the post-processing procedures, a numeric system of the verdict was used as the return of the analysis methods, which is elaborated in Table 6.8. These results were concatenated into a CSV file. This decision allows the later parts of the process to be much faster.

Table 6.8: Misuse analysis verdicts

-1	Does not include the primitive in question
0	Properly implemented
1	Dase of misuse
3	Out of scope

6.6 Metadata Extraction

As mentioned in Section 2.2, the `AndroidManifest.xml` file, among others, includes information about versioning of the apps such as app version, SDK version, and target API level. This information is particularly useful if an individual group of apps is to be analyzed with criteria other than their publication year. Notably, this research, as will be explained in Section 7.2 as part of the experimentation, aims to focus on the apps that are solely tailored for the latest version of Android.

In parallel to rule analysis, another script was deployed to read and extract the metadata for the `AndroidManifest.xml` files. The result was stored in another CSV file along with the app names to be further processed in post-processing.

6.7 Post-Processing

As previously mentioned in Section 2.2.1, some of the apps import third-party libraries that are included within the APK file, many of which contain cryptography and misuses. Thus a decision had to be made to either include or omit those libraries in the final analysis. The inclusion of these files means that the study would assess the cryptographic security of the app and, by extension, the entire app ecosystem. At the same time, its omission indicates the developer's expertise and their reference to the documentation. Therefore, we decided on the former, which arguably provides a more accurate answer to the research questions.

Subsequently, in the post-processing phase and the apps which contained at least one misuse in any of theirs files were considered insecure, any app that did not include cryptography was discarded and statistical figures were created. The following chapter elaborates on the findings of experimentation.

Chapter 7

Results and Findings

This chapter elaborates on the results of the experiment designed in Chapter 6 on real-world apps from Google Playstore. the experiment was repeated twice, first on a collection of 4900 apps that consist of 700 apps for each year since 2014. From this dataset, the apps that implement cryptography concerning the rules were further analyzed. This dataset will help understand the evolution of cryptography in the apps in parallel with the documentation. The ideal scenario would be that the apps would be consecutively more secure each year, and (2), a collection of 2000 apps developed in 2020 that are *all* compatible with the latest version of Android (i.e., Android 10, API level 29). From this dataset, the apps designed *for* Android 10 will be selected and analyzed to understand what percent of apps developed for the current Android version is following the documentation and examined their conformity with the first round of the experiment.

Section 7.1 elaborates on the first round, with an evolutionary approach, studying the trends and changes in cryptography in the apps through time. Section 7.2 discusses the second round of aiming to reinforce the first round, and observing the situation in the latest version of Android. Lastly, in Section 7.3 we present the observations made from the experiment.

7.1 Evolutionary Analysis

For the first round of the experiment, a batch of apps was randomly selected that, as mentioned previously, contains apps from 6 consecutive years, including 2020, indiscriminate of API level. This analysis aims to determine the trends and changes in the quality of apps over the years. The ideal scenario for this analysis would be that the quality of apps would increase gradually with changes in the documentation. If this is not the case, we will investigate the probable reasons behind the problem.

From the first batch of apps, on the post-processing phase, the ones that did not include related cryptography were discarded. Table 7.1 presents the exact figures. Note that from 700 apps downloaded in 2020, not all are from the latest API. These apps are merely selected based on the publication date and can vary in their API level.

Upon further investigation, in `AndroidManifest.xml` files of the decompiled apps, the API levels were extracted and presented in Table 7.2. Intriguingly, in many instances, the apps failed to correctly indicate the API level, e.g., in some cases, they did not

Table 7.1: Cryptography in apps (2014-2020)

Year	w/o crypto	with crypto
2014	168	532
2015	162	538
2016	182	518
2017	139	561
2018	182	518
2019	182	518
2020	179	521
	1194	3706
	24.36%	75.63%

Table 7.2: API level of apps

API lvl	Count	API lvl	Count	API lvl	Count
1	4	**14**	18	**23**	778
2	5	**15**	23	**24**	51
3	1	**16**	16	**25**	50
4	2	**17**	58	**26**	250
6	1	**18**	16	**27**	134
8	3	**19**	157	**28**	399
10	1	**20**	34	**29**	46
11	2	**21**	179	**N/A**	999
13	4	**22**	386	**inc**	50

mention any value or had inappropriate values (such as 1004050, which are all categorized as "inc" in the table).

7.1.1 Rule 1

The use of ECB in real circumstances, where the confidentiality of data is to be protected, is widely discouraged. Evidently, this is a well-known concept, and a substantial majority of developers do avoid its usage outside of analytical circumstances since between 50% to 80% of the total number of apps do include more proper modes; however, 14.01% of all apps released in 2020 used ECB, which in comparison to 6.56% in 2019 and 0.19% in 2018 shows a gradual increase in usage.

7.1.2 Rule 2

Rule 2 displays a contradictory result to the first, with radical changes in cases with correct implementation consistently having, as shown in Figure 7.3, less than 5% of the total. This disparity is particularly alarming when compared, Figures 7.1 and 7.3. It is revealed that many of the apps that implement the proper mode of encryption by avoiding ECB, fail to assign the IV correctly. Figure 7.4 demonstrates the overlap between the correct implementation of Rule 1 and the misuses of Rule 2. That demonstrates that even though most developers avoid ECB, they still fail to produce secure symmetric encryption. It can relate to the fact that IV has a rather complicated concept[1].

[1]Since 2014, Android cryptography implementation has drastically changed; many of the outcomes in the early years of the report have not been compatible with the experiment leading to many false

Figure 7.1: Usage trends of Rule 1

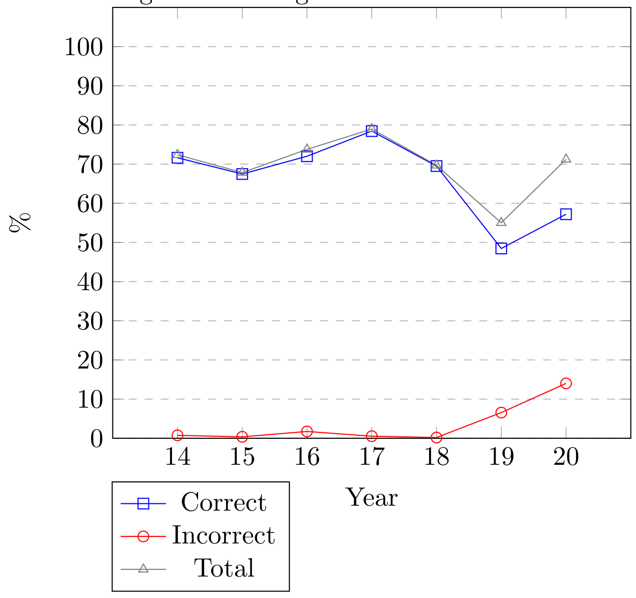
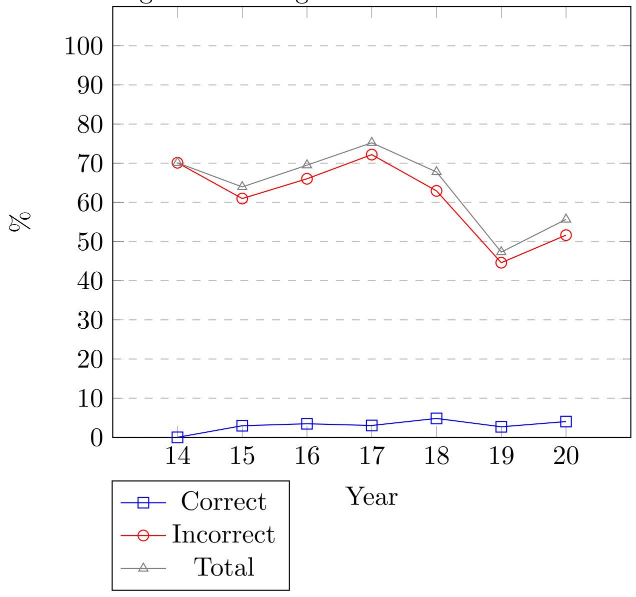


Figure 7.2: Usage trends of Rule 2



7.1.3 Rule 3

As mentioned in Chapter 5, the implementation of the key in symmetric cryptography is well elaborated by Android documentation. However, as evident in Figure 7.5, many apps fail to generate key properly. This trend has particularly worsened since 2017 and plunged to a much worse state in 2020 i.e., while between 55.2% to 78.97% of apps implement symmetric cryptography, between 16.17% and 57.01% of those have a misuse in their implementation.

positives, which were marked out of scope and disregarded in the present study.

Figure 7.3: Usage trends of Rule 2

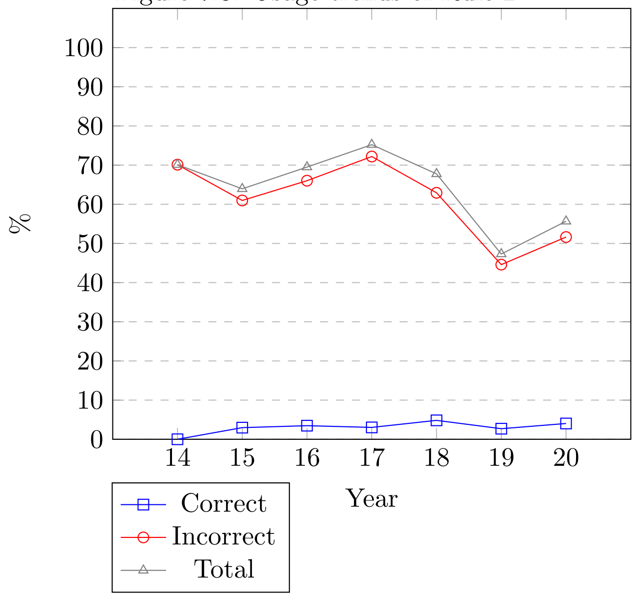
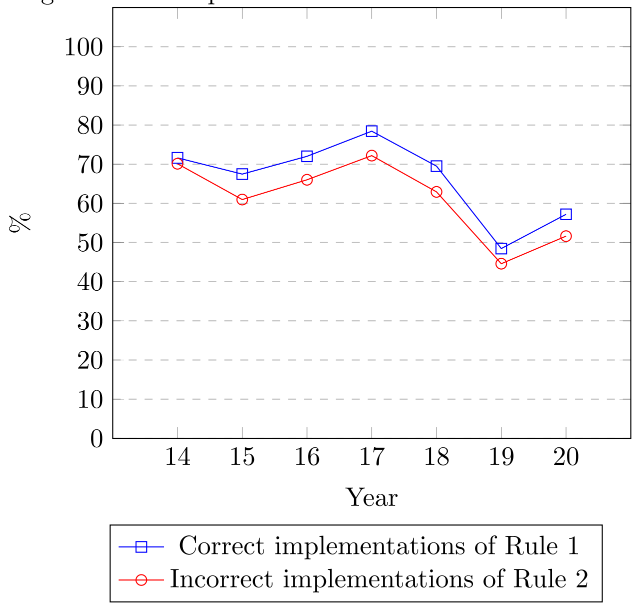


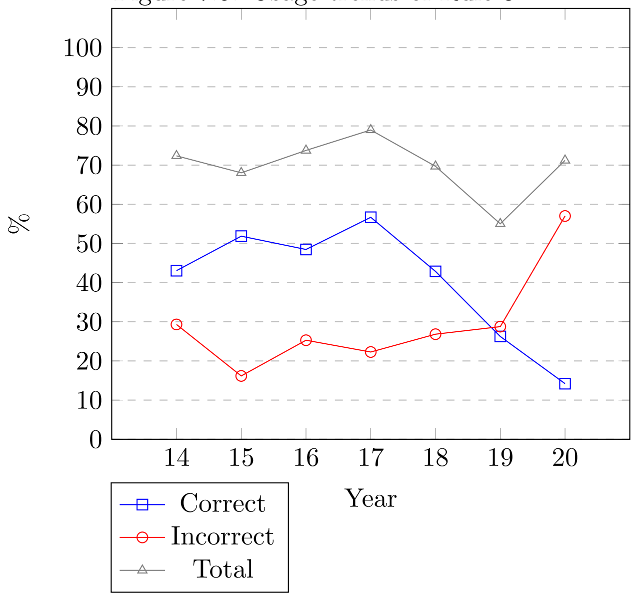
Figure 7.4: Comparison between Rule 1 and Rule 2



7.1.4 Rule 4

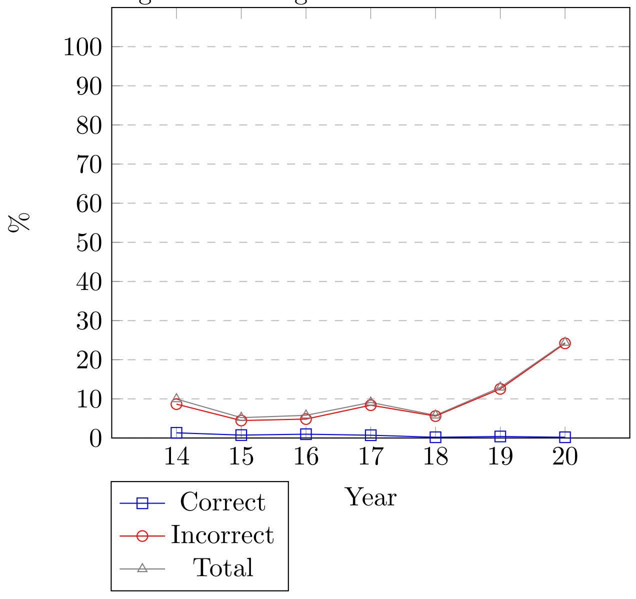
PBE, as predicted in Chapter 3, does have much fewer use cases. This small amount could mean that any developer who aims particularly to use this mechanism would want to ensure the security of their passwords and would do so with special care. Nevertheless, as can be seen in Figure 7.6, the majority have been improperly salted. In particular, the cases of the misuse from roughly 5% - 10% before 2018 has escalated to 24.18% in

Figure 7.5: Usage trends of Rule 3



2020. This change can be attributed to [62] blog, in 2016 which has an implementation of PBE.

Figure 7.6: Usage trends of Rule 4



7.1.5 Rule 5

Similar to Rule 4 discussed in the previous section, the iteration counter in PBE follows roughly the same trend as it is evident in Figure 7.7. Therefore, unsurprisingly as shown in Figure 7.8 when comparing the incorrect implementations in Figures 7.6 and 7.7 this similarity is even more apparent.

Figure 7.7: Usage trends of Rule 5

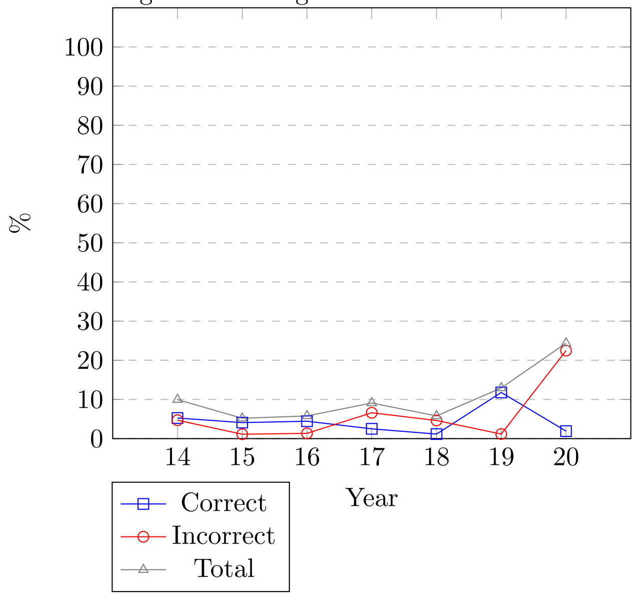
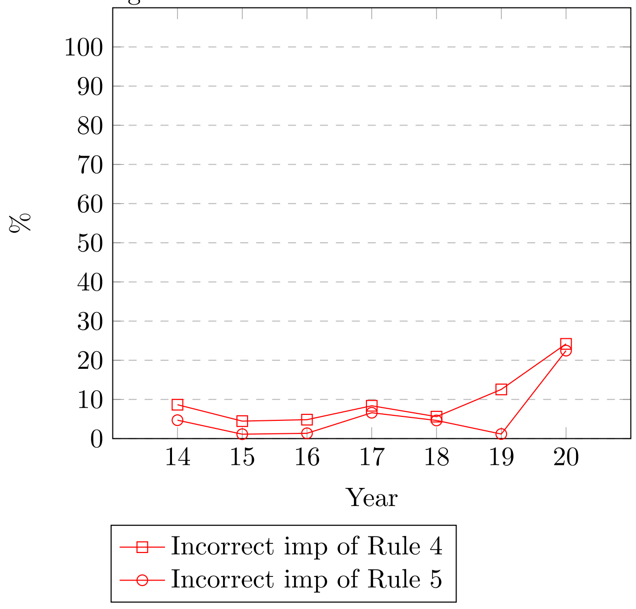


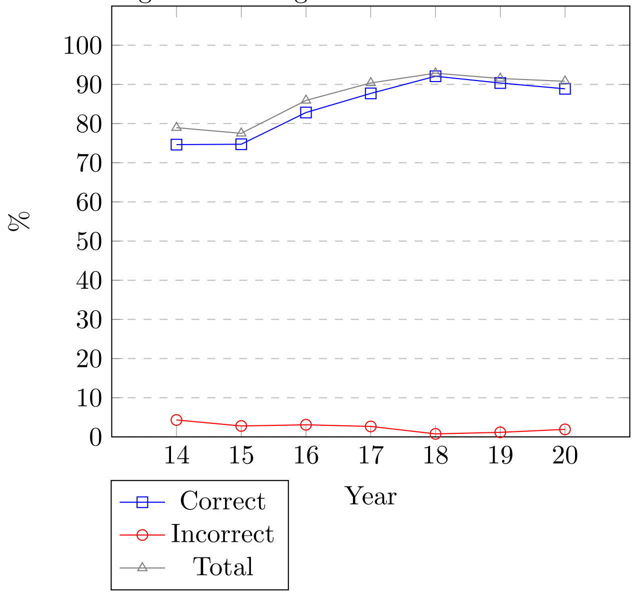
Figure 7.8: Misuses of Rules 4 and 5



7.1.6 Rule 6

Due to the peculiar nature of the 6th Rule, which requires extra steps to incorrectly be implemented, as a result of the proper default behavior, it is evident that it has much smaller misuses. As demonstrated in Figure 7.9 a vast majority of the cases are indeed correct. However, a fraction of 2% to 5% of the apps still include this misuse. This can be a good indication of how the cryptographic libraries can indeed help to create a more secure ecosystem.

Figure 7.9: Usage trends of Rule 6



7.1.7 Rule 7

According to the experiment, the 7th Rule is the most widely used and misused cryptographic primitive. As seen in Figure 7.10 while between 84.57% to 93.63% of apps implement hash algorithms, only 11.20% at most, they have implemented the primitive correctly.

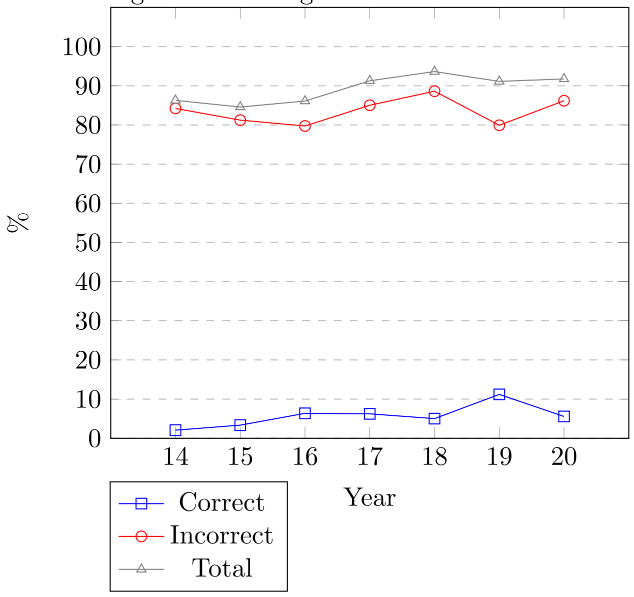
7.2 Contemporary State Analysis

In the second round of the experiment, another batch of apps published in the year 2020 and designed for the latest version of Android was randomly selected and analyzed. The reason for this experiment was that it demonstrated if the situation is different for the latest apps designed by the current version of the documentation.

For this round, from 2000 apps downloaded during pre-processing 504 were discarded as duplicates leaving 1596 apps to be decompiled and further analyzed. The reason for this relatively large number of duplicates in this batch could be that the database in 2020 was smaller (since the year was yet to pass first half at the time of writing).

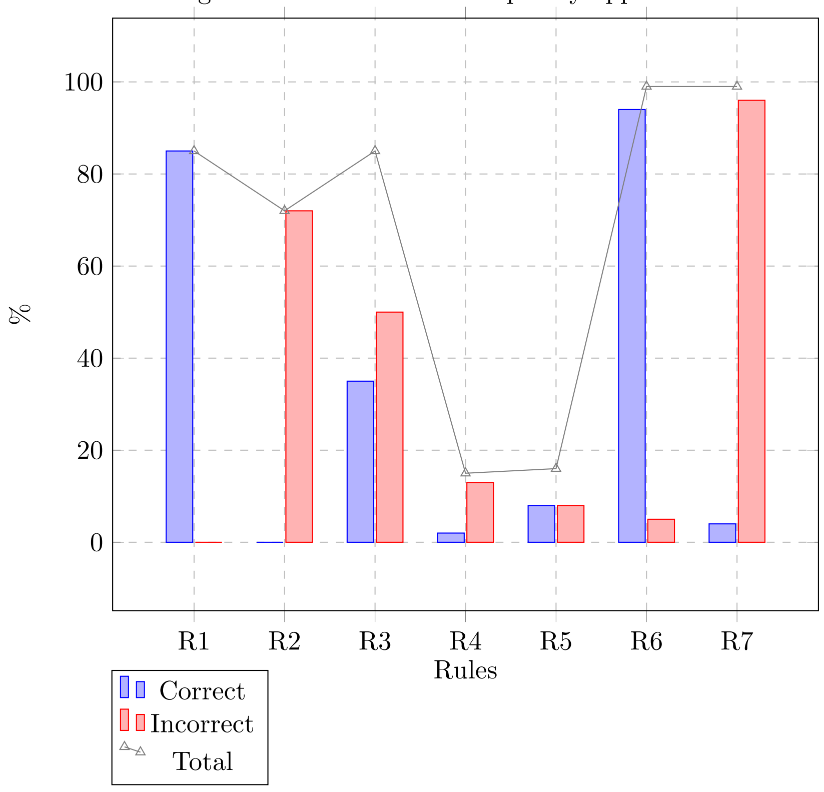
In the post-processing phase, an extra script was run, so apps that were not developed for API level 29 *and* those that did not have cryptographic primitives related to the rules were discarded. From apps that have been downloaded, only 123 or 6.15% did pass this criterion. The way the experiment was structured made all 1596 apps decompiled and analyzed which was extremely wasteful. After the analysis, the results demonstrated

Figure 7.10: Usage trends of Rule 7



in Figure 7.11 were released.

Figure 7.11: State of contemporary apps



Findings for this round confirm that it indeed follows a similar pattern to the first, demonstrating that the assumption made for the previous years and versions still stand. In addition to those assumptions, by studying Figure 7.11 the following observations can be made:

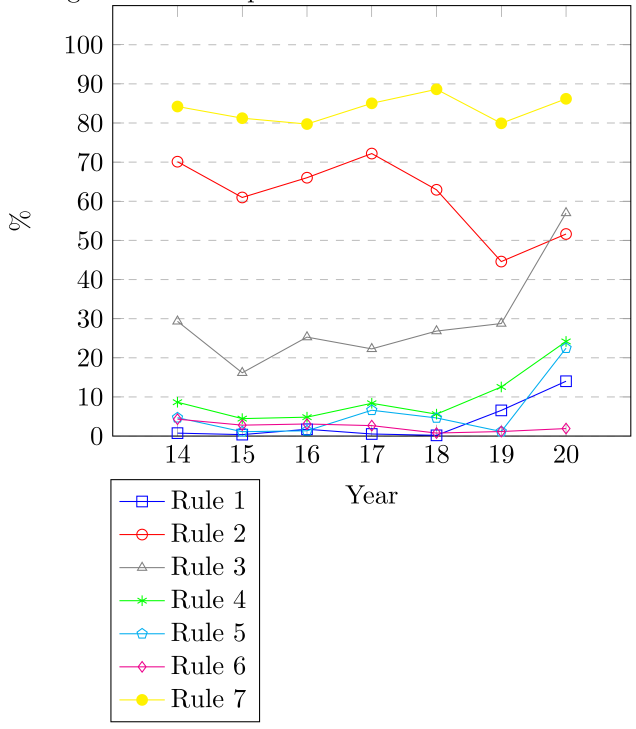
- While it is always recommended, the vast majority of developers do not follow the latest releases, as only a small number of apps released in 2020 are using the latest API version.
- The most recent version of the apps do indeed follow similar patterns as the previous round of the experiment, further reinforcing assumptions made before.
- The discrepancy between the total occurrences of the 1st and the 2nd rule, is revealed to be the use of other modes of operation such as counter mode (CTR) and similar ones which are not recommended or explored by the documentation. Hence, considered out of the scope of the experiment.

7.3 Findings

This section considers the experimentation results and provides the observations that can be made by their regard in a shortlist of items:

- As shown in Figure 7.12, although the percentage of the misuses of each rule greatly varies, they all follow a similar pattern. Namely, after starting point in 2014, between 2015 and 2016, a gradual and slight decline of misuses can be observed, while falling slightly between 2017-18, a sharp increase in recent years can be observed to occur uniformly among all rules.

Figure 7.12: Comparison of misuses of all rules



- The earliest record of all of the documentation date back to Q3 of 2018[2][3]. While this cannot be stated with complete certainty, and this trend can attribute to any number of factors, it could be related to the great changes that began in the following years. For instance, Rules 1, 3, and 4 experienced an influx of misuses immediately after these changes.

- After gradual deprication of JCPs (starting with "Crypto") in 2017 [62] and Bouncy Castle[4] in 2018 [72], there is a rather steep decline in the quantity of the apps that correctly implemented cryptography. This decline could be related to the fact that most of the JCPs enjoy extensive documentation, lacking which developers are forced to use external sources. Google announced that this decision was "because having duplicated functionality imposes additional costs and risks while not providing many benefits." [72].

- Another pitfall of Android cryptography library that particularly affects the 2nd Rule is the fact that if no IV is assigned in the call to `Cipher.getInstance(.)` a default IV of zero is assigned.

- Misuse of Hash algorithms is the most outstanding element in the graph, while they are most commonly used cryptographic primitives as well.

- Primitives with secure default behavior can dramatically reduce the number of misuses. For instance, Rule 6 in which the default behavior is secure has a very small number of misuses while Rule 2 with insecure default behavior can cause many misuses.

- As per Figures 7.3, 7.6, 7.7, and 7.10 show these rules suffer from very small percent of correct implementation (i.e. Small area between Gray and Blue lines) while 7.1 and 7.9 show promising results.

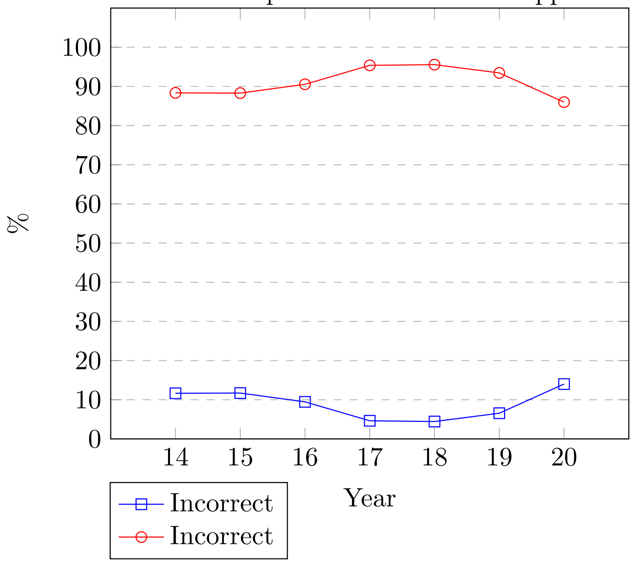
- As demonstrated in Figure 7.13, only a small percentage of apps did implement proper cryptography, i.e. from 3706 apps, 3376 had some sort of misuse embedded in their code and only 330 or 10.72% were properly implemented.

[2]Unfortunately, Android does not keep a public history of changes in the documentation, so there is no effective way to scrutinize the evolution of the documents or understand what changes in documentation exactly coincide with changes in the trends discovered by this research. The only source of the records would be the Wayback Machine archives which are widely omitted or corrupted.

[3]https://archive.org

[4]https://bouncycastle.org

Figure 7.13: Correct and incorrect implementation to total apps with cryptography



Chapter 8

Discussion

In light of the findings in Chapters 5 and 7, we revisit the research questions to state an answer within the scope of the thesis. Understandably, stating the answer to such questions requires much deliberation and care. To do so, we break them down to smaller pieces and provide the final answer as thoroughly as possible. Note that there is some overlap between the segments of the answer.

How comprehensive, usable, and reliable is the Android documentation regarding implementation of cryptographic libraries in Android apps?

- **Comprehensiveness:** In many instances, the documentation fails to remain comprehensive. For instance, as mentioned in Section 5.2, proper explanation and usage of IV is not thoroughly provided. GCM, Android's recommended mode of encryption is completely ignored. This lack of comprehension, unsurprisingly, leads to many misuses in this particular rule. There are other instances of absence of comprehensiveness in the documentation as well. As such is the explanations of hash algorithms, no information about which hash algorithms must be avoided is given, leading to a staggering number of misuses.

  It is not to say that the documentation is completely devoid of extensiveness. In some instances, such as secure random seed (concerning Rule 6), the documentation has a comprehensive approach, providing examples and feats to avoid which accompanied with secure default behavior leading to a minimal number of misuses in implementation of `SecureRandom(.)` throughout the entire years, demonstrating that with sufficient care, many of misuses can be circumvented.

- **Usability:**

  In several cases, the documentation fails to remain usable. Some of the most notable cases of which can be observed are, for instance, the message encryption algorithm is provided but, the decryption of the same algorithm is ignored (see Section 5.2). Another example is that in [62] code sample for PBE encryption has compile-time errors, or when explaining key derivation algorithms, provides samples for PBE without any prior notice or any indication. Such cases reduce the usability of the documentation and cause developers to withdraw from utilizing it altogether.

- **Reliability:** Mistakes, errors, and misleading information are occasionally scattered throughout the documentation, and this is particularly visible where the code samples include outdated encryption primitives, hardcoding of passwords, logical and syntactical errors (see 5.2 and 5.3). However, scarce instances of such mistakes are evident in the documentation.

How effective the changes in the documentation have on cryptographic correctness of apps in the Android PlayStore?

Findings of Chapter 7 and, in particular, findings of the Section 7.3 demonstrate very unpromising results for the state of Android documentation and cryptographic security, in particular against IND-CPA. Notably, as the trends show, the changes follow a trajectory soaring towards more insecure apps in the future.

Marginal Findings

Delving deep into any topic brings unexpected revelations to the surface that did not transpire at its inception, and this research is not different. In addition to the initial research questions, several other observations could be made from the research that may shed light on some of the shortcomings of the documentation:

First and foremost is that the documentation does not have a preventative approach, as common yet insecure primitives are not generally explained or prohibited, for instance, it is never mentioned that ECB, MD5 (or SHA-1) are insecure and keys should not be reused or passwords should never be hardcoded. Arguably, the documentation is not intended to replace or imitate tutorials and are not suitable for such information. Nevertheless, in other topics such as random seed, there are such measures that lead to minimal occurrences of misuse and Android developer blogs provide a suitable platform for this.

Second, the default behavior of algorithms and methods is not elaborated, to find out if this information one has to go to great length and still to ascertain this could be challenging. This data is indeed a common feature of many documentations lack of which is clearly perceived.

Chapter 9

Limitations

Due to the ubiquity of Android, its components, and resources on the web, the study of Android is very accessible. Therefore, this research enjoyed minimal limitations in its process. However, any project of this magnitude indeed has to take certain measures to be completed in a reasonable time. These limitations are manifested in this chapter.

1. Since some apps do have server-side processing that is responsible for the provision of cryptographic primitives from back-end, analyzing this particular mechanism was not possible. Therefore, there may be some false positives due to this fact.

2. Compared to the massive number of apps in the Google Playstore (2,966,572 to be exact[1]), the number of apps examined for this research (6900) seems to be a small sample space of only 0.232% of the apps, which is hardly a sufficient representation of the entire Playstore database. However, because of the hardware, time, and cost of the experiment, a sample space of this size seems reasonable.

3. The selection of the rules chosen for this thesis, in no shape or form, represent the comprehensive cryptographic security of Android apps, and they merely produce a small sample that aim to describe this concept. A more comprehensive analysis, including more extensive rules, could have a more accurate, case-by-case representation of the system's exact shortcomings. However, we chose to forgo a more comprehensive set of rules in favor of dedication of more time to each of the rules so we can have a sample that represents Google's approach to Android cryptography.

[1]Notice the disparity between this number and the one in the Introduction chapter, showing Playstore's growth for the duration of this thesis

Chapter 10

Future Work

In the ever-evolving ecosystem of mobile application security, there are many areas left to be investigated, scrutinized, and understood. As this field is a relatively new one with massive interest and sensitive data that need protection, hence, some of the recommendations that can be pointed out are listed in this section.

- iOS apps from Apple's AppStore are structured very differently since they are made with no multi-platform abilities in mind, making a similar study significantly harder. It would be intriguing to conduct a similar study on Apple to examine how that platform holds.
- Some of the reasons that such misuses happen are because Android cryptographic libraries are using conventional cryptographic schemes that may not be suitable for mobile devices as apps are quickly developed and submitted with significant impact compared with traditional software that was designed by more expert developers for a limited audience. Hence, studying the possibility of integrating more modern cryptographic mechanisms such as nonce misuse-resistance cryptography [73], may lead to significant advances in the field.
- As mentioned in the Limitations chapter, this experiment can be repeated with a sample size of magnitude larger than it could be carried out with our available resources. While to be able to reach even 1% of the total population of the apps in the Playstore, the sample size should grow 4 times or roughly 27,600 apps requiring immense resources; this could further enforce the accuracy of this study.
- While IND-CPA is an excellent property, another study that could prove useful is to examine the apps for IND-CCA to enlighten other aspects of security.
- Seven Rules that were explored in this research are indeed fundamental rules that should be followed by any app which deals with sensitive information, however, this is indeed not an exhaustive list, many more sensitive cryptographic schemes need to be investigated. Their representation in the documentation and correctness in published apps should be examined; for instance, many primitives in asymmetric encryption, including asymmetric message authentication and encryption are left to be studied.
- While finding problems in a system is comparatively easy, devising solutions to those issues is a challenging yet pleasing task of its own. Designing and maintaining libraries, writing documentation, and attending to such wide userbase requires ample resources, while a company as capable as Google should be able to accomplish

this feat, academic research can be of great help to provide eloquent remedies to the problems that are introduced in this research and can be of great worth.

Chapter 11

Conclusion

Documentation is the bridge that connects system creators to developers, one that ultimately decides and shapes how the systems are deployed, used, and abused. Their importance is ever so striking when it comes to the security and privacy of millions of unsuspecting users, such as in Android devices. This argument raises the question of how well documentation represents the topic of protecting that of their users and how effective those measures are.

To determine the answer to this question, we first had to acquaint ourselves with the Android operating system, its history, security infrastructure, how apps are constructed, and how they can be deconstructed, cryptographic baselines, and criterion in which documentations are assessed.

With the foundation of the study laid ahead, we moved to examine the related documentation in detail with careful and critical attention to expose all the critical points that can harm the security of apps. An experiment was then designed and conducted on 6900 real-world apps to illuminate the effect of the documentation on those. Lastly, the result was analyzed and cross-referenced, and it was revealed that the documentation lacks the satisfactory quality to ensure the security of their users and that many apps suffer from weak cryptographic implementation as a result.

This study is merely a testament of the work that is always needed to be done by corporations that are responsible for so much trust that has been bestowed upon them, a reminder to developers to ensure that sufficient research is crucial for creating apps that deal with peoples information, and notice for users of the dangers that such a powerful device in their possession may carry.

Bibliography

[1] D. Aamoth, *First Smartphone Turns 20: Fun Facts About Simon*, 2014. [Online]. Available: https://time.com/3137005/first-smartphone-ibm-simon/.

[2] "Smartphone Market Share", IDC, Tech. Rep., 2020. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os.

[3] *Number of Android apps on Google Play*, Tech. Rep. [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps.

[4] B. Marr, *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*, research rep., May 2018. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/.

[5] J. DeMuro, *8 reasons why smartphones are privacy nightmare*, 2018. [Online]. Available: https://www.techradar.com/news/8-reasons-why-smartphones-are-privacy-nightmare.

[6] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications", in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, New York, New York, USA: ACM Press, 2013, pp. 73–84, ISBN: 9781450324779. DOI: 10.1145/2508859.2516693. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2508859.2516693.

[7] D. Ehringer, *The dalvik virtual machine architecture*, 2010. [Online]. Available: http://davidehringer.com/software/android/The%7B%5C_%7DDalvik%7B%5C_%7DVirtual%7B%5C_%7DMachine.pdf.

[8] Q. Do, B. Martini, and K.-K. R. Choo, "Exfiltrating data from Android devices", *Computers & Security*, vol. 48, no. 2011, pp. 74–91, Feb. 2015, ISSN: 01674048. DOI: 10.1016/j.cose.2014.10.016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016740481400162X.

[9] *Codenames, Tags, and Build Numbers*, 2020. [Online]. Available: https://source.android.com/setup/start/build-numbers.

[10] *Uses-sdk*, 2020. [Online]. Available: https://developer.android.com/guide/topics/manifest/uses-sdk-element.

[11] *Supporting Older Versions*, 2020. [Online]. Available: https://source.android.com/setup/build/older-versions.

[12] *System and kernel security*, Google Inc., Jan. 2020. [Online]. Available: https://source.android.com/security/overview/kernel-security.

[13] *Kernel*, 2020. [Online]. Available: https://source.android.com/devices/architecture/kernel/.

[14] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security modules: general security support for the linux kernel", in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, IEEE, 2002, pp. 213–226, ISBN: 0-7695-2057-X. DOI: 10.1109/FITS.2003.1264934. [Online]. Available: http://ieeexplore.ieee.org/document/1264934/.

[15] J. Hoopes, *Virtualization for Security.* Elsevier, 2009, ISBN: 9781597493055. DOI: 10.1016/B978-1-59749-305-5.X0001-1. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9781597493055X00011.

[16] L. Vokorokos, A. Baláž, and B. Madoš, "Application Security through Sandbox Virtualization", *Acta Polytechnica Hungarica*, vol. 12, no. 1, pp. 83–101, Feb. 2014, ISSN: 17858860. DOI: 10.12700/APH.12.1.2015.1.6. [Online]. Available: http://uni-obuda.hu/journal/Vokorokos%7B%5C_%7DBalaz%7B%5C_%7DMados%7B%5C_%7D57.pdf.

[17] W. B. Tesfay, T. Booth, and K. Andersson, "Reputation Based Security Model for Android Applications", in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, Jun. 2012, pp. 896–901, ISBN: 978-1-4673-2172-3. DOI: 10.1109/TrustCom.2012.236. [Online]. Available: http://ieeexplore.ieee.org/document/6296066/.

[18] *Verified Boot*, 2020. [Online]. Available: https://source.android.com/security/verifiedboot/.

[19] *Open APK File*, 2018. [Online]. Available: https://openapkfile.com.

[20] G. Nolan, *Decompiling Android*, J. Markham, Ed. Berkeley, CA: Apress, 2012, p. 304, ISBN: 978-1-4302-4248-2. DOI: 10.1007/978-1-4302-4249-9. [Online]. Available: http://link.springer.com/10.1007/978-1-4302-4249-9.

[21] *Kotlin FAQ*, Jet Brains, Mar. 2020. [Online]. Available: https://kotlinlang.org/docs/reference/faq.html.

[22] J. Katz, *Introduction to Modern Cryptography*, Second Edi. Chapman and Hall/CRC, Aug. 2007, vol. 1, ISBN: 9781420010756. DOI: 10.1201/9781420010756. [Online]. Available: https://www.researchgate.net/publication/220688729%7B%5C_%7DIntroduction%7B%5C_%7Dto%7B%5C_%7DModern%7B%5C_%7DCryptography.

[23] Bellare, "A Note on Negligible Functions", *Journal of Cryptology*, vol. 15, no. 4, pp. 271–284, Sep. 2002, ISSN: 0933-2790. DOI: 10.1007/s00145-002-0116-x. [Online]. Available: http://link.springer.com/10.1007/s00145-002-0116-x.

[24] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., Mar. 2001, p. 1..28. DOI: 10.6028/NIST.SP.800-38G. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[25] M. Bellare, J. Kilian, and P. Rogaway, "The Security of Cipher Block Chaining", in *Advances in Cryptology — CRYPTO '94*, vol. 839 LNCS, Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 341–358, ISBN: 9783540583332. DOI: 10.1007/3-540-48658-5_32. [Online]. Available: http://link.springer.com/10.1007/3-540-48658-5%7B%5C_%7D32.

[26] H. Heys, "Analysis of the statistical cipher feedback mode of block ciphers", *IEEE Transactions on Computers*, vol. 52, no. 1, pp. 77–92, Jan. 2003, ISSN: 0018-9340. DOI: 10.1109/TC.2003.1159755. [Online]. Available: http://ieeexplore.ieee.org/document/1159755/.

[27] D. a. McGrew, "Counter Mode Security : Analysis and Recommendations", pp. 1–8, 2002. [Online]. Available: http://cr.yp.to/bib/2002/mcgrew.pdf.

[28] P. Rogaway, “Nonce-Based Symmetric Encryption”, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3017, 2004, pp. 348–358. DOI: 10.1007/978-3-540-25937-4_22. [Online]. Available: http://link.springer.com/10.1007/978-3-540-25937-4%7B%5C_%7D22.

[29] H. Feistel, *Cryptography and Computer Privacy*, 5. Scientific American, May 1973, vol. 228, ch. 5, pp. 15–23. [Online]. Available: http://www.apprendre-en-ligne.net/crypto/bibliotheque/feistel/.

[30] C. H. W. Eyer, J. L. Smith, and W. L. Tuchman, *Essage Verification and Transmission Error Detection By Block Chaining*, 1978. [Online]. Available: https://patents.google.com/patent/US4074066A/en.

[31] R. Munir, “Security analysis of selective image encryption algorithm based on chaos and CBC-like mode”, in *2012 7th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*, IEEE, Oct. 2012, pp. 142–146, ISBN: 978-1-4673-4550-7. DOI: 10.1109/TSSA.2012.6366039. [Online]. Available: http://ieeexplore.ieee.org/document/6366039/.

[32] S. Lemsitzer, J. Wolkerstorfer, N. Felber, and M. Braendli, “Multi-gigabit GCM-AES Architecture Optimized for FPGAs”, in *Cryptographic Hardware and Embedded Systems - CHES 2007*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 227–238. DOI: 10.1007/978-3-540-74735-2_16. [Online]. Available: http://link.springer.com/10.1007/978-3-540-74735-2%7B%5C_%7D16.

[33] D. A. Mcgrew and J. Viega, *The Galois / Counter Mode of Operation (GCM) Intellectual Property Statement*, 2004. [Online]. Available: https://regmedia.co.uk/2017/01/12/gcm-spec.pdf.

[34] National Institute of Standards and Technology, “Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program National Institute of Standards and Technology”, National Institute of Standards and Technology, Tech. Rep., 2019, pp. 1–63. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Module-Validation-Program/documents/fips140-2/FIPS1402IG.pdf.

[35] C. E. Shannon, “Communication Theory of Secrecy Systems”, *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, Oct. 1949, ISSN: 00058580. DOI: 10.1002/j.1538-7305.1949.tb00928.x. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6769090.

[36] A. Shamir, “New Directions in Croptography”, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. TT22, 2001, pp. 159–159, ISBN: 3540425217. DOI: 10.1007/3-540-44709-1_14. [Online]. Available: http://link.springer.com/10.1007/3-540-44709-1%7B%5C_%7D14.

[37] R. Morris and K. Thompson, “Password security: a case history”, *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, Nov. 1979, ISSN: 00010782. DOI: 10.1145/359168.359172. [Online]. Available: http://portal.acm.org/citation.cfm?doid=359168.359172.

[38] V. Goyal, V. Kumar, M. Singh, A. Abraham, and S. Sanyal, “CompChall: addressing password guessing attacks”, in *International Conference on Information Technology: Coding and Computing (ITCC’05) - Volume II*, vol. 1, IEEE, 2005, 739–744 Vol. 1, ISBN: 0-7695-2315-3. DOI: 10.1109/ITCC.2005.107. [Online]. Available: http://ieeexplore.ieee.org/document/1428552/.

[39] P. Oechslin, “Making a Faster Cryptanalytic Time-Memory Trade-Off”, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2729, Berlin, Heidelberg: Springer, 2003, pp. 617–630. DOI: 10.1007/978-3-540-45146-4_36. [Online]. Available: http://link.springer.com/10.1007/978-3-540-45146-4%7B%5C_%7D36.

[40] K. M. Moriarty, B. Kaliski, and A. Rusch, *PKCS #5: Password-Based Cryptography Specification Version 2.1*, 2017. DOI: 10.17487/RFC8018. [Online]. Available: https://www.rfc-editor.org/info/rfc8018.

[41] B. Jun and P. Kocher, “The Intel® Random Number Generator”, *The Bell System Technical Journal*, vol. 27, no. July 1948, pp. 379–423, 1999. [Online]. Available: http://www.csshl.net/sites/default/files/downloadable/crypto/intelRNG.pdf.

[42] C. Shannon, “A Mathematical Theory of Communication”, *The Bell System Technical Journal*, vol. 27, no. 4, pp. 379–423, 1948, ISSN: 00160032. DOI: 10.1016/s0016-0032(23)90506-5. [Online]. Available: http://people.math.harvard.edu/%7B~%7Dctm/home/text/others/shannon/entropy/entropy.pdf.

[43] A. Figotin, I. Vitebskiy, V. Popovich, G. Stetsenko, S. Molchanov, A. Gordon, J. Quinn, and N. Stavrakas, *Random number generator based on the spontaneous alpha-decay*, 2002. [Online]. Available: https://patentimages.storage.googleapis.com/57/f5/af/22c880b4ce6880/US6745217.pdf.

[44] Q. Zhou, X. Liao, K.-w. Wong, Y. Hu, and D. Xiao, “True random number generator based on mouse movement and chaotic hash function”, *Information Sciences*, vol. 179, no. 19, pp. 3442–3450, Sep. 2009, ISSN: 00200255. DOI: 10.1016/j.ins.2009.06.005. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0020025509002540.

[45] Y. Hu, X. Liao, K.-w. Wong, and Q. Zhou, “A true random number generator based on mouse movement and chaotic cryptography”, *Chaos, Solitons & Fractals*, vol. 40, no. 5, pp. 2286–2293, Jun. 2009, ISSN: 09600779. DOI: 10.1016/j.chaos.2007.10.022. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0960077907008958.

[46] B. Preneel, “Analysis and Design of Cryptographic Hash Functions”, *Angewandte Chemie International Edition*, vol. 40, no. 6, p. 9823, Mar. 2001, ISSN: 14337851. DOI: 10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C. [Online]. Available: https://www.academia.edu/36857585/Analysis%7B%5C_%7Dand%7B%5C_%7DDesign%7B%5C_%7Dof%7B%5C_%7DSymmetric%7B%5C_%7DCryptographic%7B%5C_%7DAlgorithms.

[47] X. Wang and H. Yu, “How to Break MD5 and Other Hash Functions”, in *Lecture Notes in Computer Science*, vol. 3494, 2005, pp. 19–35. DOI: 10.1007/11426639_2. [Online]. Available: http://link.springer.com/10.1007/11426639%7B%5C_%7D2.

[48] M. Weiser, “Program Slicing”, *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984, ISSN: 0098-5589. DOI: 10.1109/TSE.1984.5010248. [Online]. Available: http://ieeexplore.ieee.org/document/5010248/.

[49] R. Gold, “Control flow graphs and code coverage”, *International Journal of Applied Mathematics and Computer Science*, vol. 20, no. 4, pp. 739–749, Dec. 2010, ISSN: 1641-876X. DOI: 10.2478/v10006-010-0056-9. [Online]. Available: http://content.sciendo.com/view/journals/amcs/20/4/article-p739.xml.

[50] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, “Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications”, in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, Beijing: IEEE, Aug. 2014, pp. 75–80, ISBN: 978-1-4799-5079-9. DOI: 10.1109/DASC.2014.22. [Online]. Available: https://ieeexplore.ieee.org/document/6945307/.

[51] D. Callahan, A. Carle, M. Hall, and K. Kennedy, “Constructing the procedure call multigraph”, *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 483–487, Apr. 1990, ISSN: 00985589. DOI: 10.1109/32.54302. [Online]. Available: http://ieeexplore.ieee.org/document/54302/.

[52] A. Chatzikonstantinou, C. Ntantogian, G. Karopoulos, and C. Xenakis, “Evaluation of Cryptography Usage in Android Applications”, in *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, Athens: ACM, 2016, ISBN: 978-1-63190-100-3. DOI: 10.4108/eai.3-12-2015.2262471. [Online]. Available: http://eudl.eu/doi/10.4108/eai.3-12-2015.2262471.

[53] S. Ma, D. Lo, T. Li, and R. H. Deng, “CDRep: Automatic Repair of Cryptographic Misuses in Android Applications Siqi”, in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS ’16*, Singapore, Singapore: ACM Press, 2016, pp. 711–722, ISBN: 9781450342339. DOI: 10.1145/2897845.2897896. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2897845.2897896.

[54] I. Muslukhov, Y. Boshmaf, and K. Beznosov, “Source attribution of cryptographic API misuse in android applications”, in *ASIACCS 2018 - Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security*, New York, New York, USA: ACM Press, 2018, pp. 133–146, ISBN: 9781450355766. DOI: 10.1145/3196494.3196538. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3196494.3196538.

[55] *Cipher*, Google Inc., Google Inc., Dec. 2019. [Online]. Available: https://developer.android.com/reference/javax/crypto/Cipher.

[56] *Cryptography*, Google inc., Jan. 2020. [Online]. Available: https://developer.android.com/guide/topics/security/cryptography.

[57] *Migrating Apps to Android 9*, Google Inc., Feb. 2020. [Online]. Available: https://developer.android.com/about/versions/pie/android-9.0-migration.

[58] *KeyStore*, Google Inc., Dec. 2019. [Online]. Available: https://developer.android.com/reference/java/security/KeyStore.

[59] *Android keystore system*, Google Inc., Dec. 2019. [Online]. Available: https://developer.android.com/training/articles/keystore.

[60] *KeyGenerator*, Google Inc., Dec. 2019. [Online]. Available: https://developer.android.com/reference/javax/crypto/KeyGenerator.

[61] T. Johns, *Using Cryptography to Store Credentials Safely*, Feb. 2013. [Online]. Available: https://android-developers.googleblog.com/2013/02/using-cryptography-to-store-credentials.html.

[62] S. Giro, *Security ”Crypto” provider deprecated in Android N*, Google Inc., Jun. 2016. [Online]. Available: https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html.

[63] *PBEParameterSpec*, Google Inc., Dec. 2019. [Online]. Available: https://developer.android.com/reference/javax/crypto/spec/PBEParameterSpec.

[64] *Android Security Vulnerability*, Bitcoin.org, Aug. 2013. [Online]. Available: https://bitcoin.org/en/alert/2013-08-11-android.

[65] A. Klyubin, *Some SecureRandom Thoughts*, Google Inc., Aug. 2013. [Online]. Available: https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html.

[66] F. Chung, *Security Enhancements in Jelly Bean*, Google inc., 2014. [Online]. Available: https://android-developers.googleblog.com/2013/02/security-enhancements-in-jelly-bean.html.

[67] Y. Wang and T. Nicol, "On statistical distance based testing of pseudo random sequences and experiments with PHP and Debian OpenSSL", *Computers & Security*, vol. 53, pp. 44–64, Sep. 2015, ISSN: 01674048. DOI: 10.1016/j.cose.2015.05.005. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167404815000693.

[68] S.-j. Chang, R. Perlner, W. E. Burr, M. Sonmez Turan, J. M. Kelsey, S. Paul, and L. E. Bassham, "Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition", National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., Nov. 2012. DOI: 10.6028/NIST.IR.7896. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf.

[69] *MessageDigest*, Google Inc., Dec. 2019. [Online]. Available: https://developer.android.com/reference/java/security/MessageDigest.

[70] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community", in *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, New York, New York, USA: ACM Press, 2016, pp. 468–471, ISBN: 9781450341868. DOI: 10.1145/2901739.2903508. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2901739.2903508.

[71] K. Thompson, "Programming Techniques: Regular expression search algorithm", *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, Jun. 1968, ISSN: 00010782. DOI: 10.1145/363347.363387. [Online]. Available: http://portal.acm.org/citation.cfm?doid=363347.363387.

[72] A. Vartanian, *Cryptography Changes in Android P*, 2018. [Online]. Available: https://android-developers.googleblog.com/2018/03/cryptography-changes-in-android-p.html.

[73] S. Gueron, A. Langley, and Y. Lindell, "AES-GCM-SIV: Nonce misuse-resistant authenticated encryption", *posted at https://tools. ietf. org/html/draft-irtf-cfrg-gcmsiv-02*, vol. 26, 2016. [Online]. Available: https://www.hjp.at/doc/rfc/rfc8452.html.

Appendices

Appendix A

Experimentation Output

This appendix presents the output of the experiment, the graphs of which were introduced in Chapter 7 separately for Evolutionary Analysis and the Contemporary Analysis.

A.1 Contemporary Analysis

Table A.1: Contemporary analysis numbers

Year	**2020**		
Total	1596	**Rule 4 Correct**	2
Faulty	106	**Rule 4 Incorrect**	14
Rule 1 Correct	90	**Rule 5 Correct**	8
Rule 1 Incorrect	0	**Rule 5 Incorrect**	8
Rule 2 Correct	0	**Rule 6 Correct**	100
Rule 2 Incorrect	76	**Rule 6 Incorrect**	5
Rule 3 Correct	37	**Rule 7 Correct**	4
Rule 3 Incorrect	53	**Rule 7 Incorrect**	102

A.2 Evolutionary Analysis

Table A.2: Evolutionary analysis numbers

Year	**2020**	**2019**	**2018**	**2017**	**2016**	**2015**	**2014**
Total	521	518	518	561	518	538	532
Faulty	448	484	495	535	469	475	470
Rule 1 Correct	298	251	360	440	373	363	381
Rule 1 Incorrect	73	34	1	3	9	2	4
Rule 2 Correct	21	14	25	17	18	16	0
Rule 2 Incorrect	269	231	326	405	342	328	373
Rule 3 Correct	74	136	222	318	251	279	229
Rule 3 Incorrect	297	149	139	125	131	87	156
Rule 4 Correct	1	2	1	4	5	4	7
Rule 4 Incorrect	126	65	29	47	25	24	46
Rule 5 Correct	10	61	6	14	23	22	28
Rule 5 Incorrect	117	6	24	37	7	6	25
Rule 6 Correct	463	468	477	492	429	402	397
Rule 6 Incorrect	10	6	4	15	16	15	23
Rule 7 Correct	29	58	26	35	33	18	11
Rule 7 Incorrect	449	414	459	477	413	437	448

Total is the number of apps after post processing and Faulty is the number of apps with at least one misuse.