An Empirical Evaluation of Cryptography Usage in Android Applications

Alexia Chatzikonstantinou
Mezza Group
ahatzikostantinou@imc.com.gr

Christoforos Ntantogian
Department of Digital Systems, University of Piraeus
dadoyan@unipi.gr

Georgios Karopoulos
Department of Informatics and Telecommunications, University of Athens
gkarop@di.uoa.gr

Christos Xenakis
Department of Digital Systems, University of Piraeus
xenakis@unipi.gr

ABSTRACT
Mobile application developers are using cryptography in their products to protect sensitive data like passwords, short messages, documents etc. In this paper, we study whether cryptography and related techniques are employed in a proper way, in order to protect these private data. To this end, we downloaded 49 Android applications from the Google Play marketplace and performed static and dynamic analysis in an attempt to detect possible cryptographic misuses. The results showed that 87.8% of the applications present some kind of misuse, while for the rest of them no cryptography usage was detected during the analysis. Finally, we suggest countermeasures, mainly intended for developers, to alleviate the issues identified by the analysis.

Categories and Subject Descriptors
E.3 [**Data**]: Data Encryption - *code breaking, data encryption standard (DES), public key cryptosystems, standards (e.g., DES, PGP, RSA)*

General Terms
Design, Experimentation, Security

Keywords
Software security, Android, Cryptography misuse

1. INTRODUCTION
The need to privately share information in a manner that would be understandable to only a specific group of people exists for thousands of years before computer's invention and establishment. The existence of cryptographic algorithms akin to Caesar's Cipher proves that contemporary cryptography has its origins in Caesar's era, when attempts to achieve information security began to take place. Thus, the field of cryptography is not new and efforts towards its improvement exist for many years.

The rapid technological progress in the last years has led to the emergence of smartphones which, apart from voice and SMS, support Internet access, standalone applications, and wireless connectivity. The same devices are used by a large proportion of users to install applications that store sensitive data like passwords, location, and social network interactions.

The need for privacy imposes cryptography utilization in applications that manage these sensitive data. To this end developers embed cryptographic techniques in their mobile applications; and while cryptography is a long existing field, developers rarely have knowledge of information security. As a consequence, incidents of data breaching and disclosure are very frequent, while there are even stories of popular products that utilize practically no security; a recent infamous example is NQ Mobile Vault[1] application, which was discovered that it uses a simple XOR function to perform cryptography.

Regarding the academic activity in the specific domain, a lot of research has been conducted and many studies have been realized; however, none of them has yet concentrated on a set of good and bad practices, as each work aims at giving prominence to the specific cryptographic mistakes of the applications and not at developers training. Our contributions, in this paper, are: (a) to evaluate the use of cryptographic techniques in real world Android applications and feature the most common misuses, and (b) to provide a list of good practices for developers in order to alleviate the identified issues. The reason we focus on Android is because it is one of the prominent smartphone platforms with a relatively stable cryptographic API (Java's Cipher), and has numerous applications available.

Our approach regarding application analysis was to employ a combination of both techniques of static and dynamic analysis, so as to succeed in producing more accurate results. Generally, the term *Static Analysis* refers to the process of detecting software errors and defects or security flaws by examining the source code of a program without executing it, and can also be utilized to ensure conformance with specific programming requirements. Static Analysis is considered as a part of Code Review process and provides better perception of code structure [10]. Developers

Conference'10, Month 1–2, 2010, City, State, Country.

[1] http://www.slate.com/articles/technology/bitwise/2015/04/nq_mobile_vault_the_popular_encryption_app_has_laughably_crackable_encryption.html

frequently perform static analysis combining automated tools and visual source code inspection[2].

On the other hand, *Dynamic Analysis* refers to the testing and evaluation of a program based on its execution and it is usually performed with a view to detecting subtle defects or vulnerabilities manifested during runtime, the cause of which is too perplex to be detected via static analysis[3]. Developers, through a dynamic test, are capable of monitoring system memory, functional behavior, response time, and overall performance of the system[4]. Therefore, there are cases where a single component from the abovementioned list is selected to be examined (e.g. system memory) in order to seek only for specific types of errors.

Regarding the advantages of the two methods, Static Analysis is the most thorough technique and the developers using it are capable of identifying the exact location of weaknesses in the code, as well as of examining all possible execution paths and variable values and not just those invoked during execution. Moreover, Static Analysis reveals errors in the initial stages of the development life cycle, reducing the cost to fix and preventing errors from manifesting themselves and triggering any incident. Dynamic Analysis is more flexible regarding the possibility to test the application for apropos specified error categories only, for instance security flaws. What is more, via Dynamic Analysis it is technically feasible to test applications even if there is no access to their source code. Finally, Dynamic Analysis can be utilized as a validation of Static Analysis results.

Nevertheless, the two methods of analysis have many disadvantages both due to their nature per se, but also due to the fact that the use of automated tools for analysis is widespread. In cases where automated tools are utilized, the significant number of false positives and false negatives constitute the main drawback in both types of analysis as the tools' efficiency is highly dependent on the rules defined for software scanning. This specific fact remarks the necessity for the human factor involvement for understanding whether the tool alerted a real error or not. Additionally, Static Analysis cannot provide satisfactory results regarding memory leaks and concurrency errors. In order to detect this type of faults it is necessary to execute the software. Lastly, when Static Analysis is performed by a tool, there is a limitation regarding the programming languages that can be supported. Consequently, we can deduce that the two approaches are complementary as no single approach can find every possible type of error. Moreover, taking into account automated tools' inefficiencies, we have chosen to use manual static analysis in combination with dynamic analysis, so as to have more accurate results.

Using a combination of static and dynamic analysis, we evaluated a total of 49 Android applications downloaded from the Google Play marketplace. Our overall results feature that 87.8% of the applications show evidence of cryptography misuse, while for the rest 12.2% no cryptography was detected from our analyses. This high proportion of misuse amplifies our previous argument that developers rarely understand how to incorporate cryptography in their applications effectively.

The rest of our paper is organized as follows. Section 2 briefly presents important cryptographic concepts, while section 3 analyzes the related work. Section 4 elaborates on a set of cryptographic weaknesses that we will be used to evaluate the cryptographic security of the examined applications. Section 5 analyzes the carried out experiments by presenting the methodology for static and dynamic analysis. Section 6 evaluates the cryptographic security of the mobile application by analyzing the numerical results, while section 7 concludes the article.

2. CRYPTOGRAPHIC CONCEPTS

The key goal of encryption is to provide confidentiality and privacy; nonetheless, applications which employ cryptography can be attacked in many different ways. The most usual way is breaking encryption schemes incorporated in the application.

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

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

A secure cryptosystem constitutes any entity employing cryptography, in hardware or software level, which, given the ciphertext, averts the threat of an adversary to discern even a single bit of information describing the plaintext in polynomial time. Taking this into consideration, we should only consider an encryption scheme to be secure if and only if it is IND-CPA secure. Moreover, an encryption scheme must be either probabilistic or stateful to be IND-CPA secure [2]. Otherwise, the adversary will be able to discern if the same message was sent twice. It is noted that in a stateful encryption scheme the keys are updated in each encryption, while in a probabilistic encryption scheme randomness is used in the encryption algorithm which satisfies collision resistance and hides all the information related to its input [5].

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

3. RELATED WORK

This section provides an overview of previous work realized in static analysis, dynamic analysis, and techniques for combined static and dynamic analysis.

The first methodical attempt that constitutes a key milestone in the specific domain is Manuel Egele's *et al.* study [4], the main purpose of which was to test whether developers use the cryptographic APIs in a fashion that provides typical

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

[3] http://gcn.com/articles/2009/02/09/static-vs-dynamic-code-analysis.aspx

[4] https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing

cryptographic notions of security (e.g. IND-CPA security). Their system, namely CryptoLint, uses static program slicing and analyzes compiled Android applications having no access to the source code. The results showed that 88% of applications that use cryptographic APIs make at least one mistake.

One of the drawbacks of this approach is that the tool is not open source so it is not possible to repeat the experiments. Moreover, the list of checked applications is not available. Also, CryptoLint lacks the capability of analyzing cryptographic primitives' invocation from native code (i.e. code written in other language than Java, for example C and C++), as its functionality focuses on Dalvik bytecode investigation. CryptoLint also does not include the identification of all types of non-predictable IVs, as the static IV's recognized by the tool refer to a subcategory of non-predictable IVs. A general drawback of automated tools is false alarms[5]. Thus, manual static analysis seems to be a more proper approach, guaranteeing more accurate results as well as the ability to cover a greater extent of cryptographic rules.

Yong Li *et al.* introduced iCryptoTracer [9], a tool similar to CryptoLint, though its function is based on a combination of both static and dynamic analysis techniques and its focus is on iOS applications. This tool first uses static analysis to scan and record the APIs' locations of cryptographic functions. Then, during the dynamic analysis phase, it monitors those API calls at runtime. Finally, iCryptoTracer, combining the information gathered on the previous steps with its diagnosis engine, decides whether a cryptographic misuse exists or not in the application. The results showed that approximately 65.3% of the applications examined contain various degrees of security flaws caused by cryptographic misuse. The main drawback of this method is that an insufficient set of rules is provided, according to which applications are classified into Healthy, Weak or Critical.

A quite similar study has been also conducted by Somak Das *et al.* [3], who systematically compared the APIs of cryptographic libraries across different programming languages (C, C++, Java, Python and Go) and evaluated their potential for misuse. In this report the possibility to have data security breaches is considered irrespective of the security of cryptography library in use, and it depends on the manner that the developer uses the library and consequently, on the properties of each particular library that encourage or discourage cryptographic misuse.

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

A literature review of cryptography on Android message applications has been presented by Nishika and Rahul Kumar Yadav [13], who surveyed and illustrated the most common and widely used SMS encryption techniques, inferring that there is a need for an efficient encryption algorithm.

The most recent work in this field of study is that realized by Shuai *et al.* [18]. In their study, the authors initially define specific models of cryptographic misuse, in which they are based so as to build a tool of auto detection (CMA). CMA employs both static and dynamic analysis techniques in order to detect cryptographic vulnerabilities and it is tested in 45 Android applications downloaded from the Chinese application store Baidu. However, CMA misses cases where cryptography is employed but is not included in the specific API (when, for example, the developer has implemented a custom cryptographic algorithm). This fact also indicates the need for including more models for cryptographic misuse in the list. CMA's paper includes a quite satisfying number of cryptographic primitives that have to be taken into consideration in such an analysis, which is something that similar papers lack. Nevertheless, the tool created is not designed to locate all the models of cryptographic misuse mentioned in the paper, as for example the key management category of flaws is omitted. Additionally, there are models that, according to the results, are not violated by any of the applications under examination, which makes the proper functionality of the tool for the specific models and the necessity of the specific models doubtful. As a result, there are only results for the trivial cryptographic principles misuses. Last but not least, it has to be remarked the fact that the applications were not downloaded from the official Android marketplace but instead they used the Chinese application store Baidu.

The majority of the related works are based on automated static or dynamic analysis tools; however, although automated tools offer the advantage of being able to examine a large number of applications, it is always possible to miss certain types of flaws. Moreover, automated static analysis tools have proven to generate a fair number of false positives while in manual static analysis the findings can be verified. Taking into account automated tools' inefficiencies, we have chosen to use manual static analysis in combination with dynamic analysis, so as to have more accurate results. Our purpose is to cover a detailed list of cryptographic flaws and misuses, something that developers' community lacks, with a view to helping programmers avoid common cryptographic misuses.

4. CRYPTOGRAPHIC WEAKNESSES

In this section, we evaluate the cryptographic security of the examined applications. To this end, we classify and analyze cryptographic weaknesses using four categories: (a) use of weak cryptography, (b) weak implementations, (c) use of weak keys, and (d) use of weak cryptographic parameters.

Weak cryptography. This category comprises cryptographic algorithms that are used in applications despite the fact that it is well known that they are not secure.

C1. **Use of weak cryptographic algorithms or hash functions**. Programmers should not use algorithms proven to be broken or weak. For example, MD4, MD5, SHA1, DES and RC4 are considered to be obsolete [8].

C2. **Use of custom cryptographic algorithms**. The security offered by non-publicly reviewed algorithms invented by programmers themselves is questionable and their employment is considered to be insecure [17].

C3. **Use of cryptographic algorithms in ECB mode.** It does not constitute a secure cryptographic mode, as it cannot be IND-CPA secure [4].

[5] www.owasp.org/index.php/Static_Code_Analysis

C4. Use of non-Cryptographically Secure PseudoRandom Number Generators (CSPRNGs). CSPRNGs seed data with the required entropy in order to make it much more difficult for adversaries to guess the produced random numbers [21]. The factor of randomness should also be introduced in any kind of password, salt and seed. Java provides for Android Development the SecureRandom class which implements a PseudoRandom Number Generator (PRNG) for keys production [8, 17]; the Random class, however, is not considered secure and should not be used for key generation.

C5. Use of CBC combined with PKCS5Padding. This mode is vulnerable to padding oracle attacks, while PKCS7Padding is considered to be the best option for the specific encryption mode [1, 6, 7, 15, 16, 19, 20].

C6. No cryptography usage observed. This weakness comprises the cases where no cryptographic operation was identified during the static and dynamic analysis. This includes cases where either obscure cryptography is used or no cryptography is used at all.

Weak implementations. The utilization or implementation of cryptographic algorithms in a non-standard manner or not following best practices can result in unsafe applications.

I1. Re-implementing standard algorithms (e.g. AES). Re-implementations of well-known algorithms are also possible to be incorrect and insecure. Thus, developers should not use other than well-known cryptographic algorithm implementations [17].

I2. Use of PBE with no salt. It is recommended to use PBE with random salts in order to avoid brute force attacks [4].

I3. Use of PBE with fewer than 1,000 iterations. This should also be avoided in order to prevent brute force attacks [4].

I4. Use of static or reuse of PRNG seed. A PRNG seed must not be reused in the same context as it is a best practice to use independent random numbers in all stages of a cryptographic procedure. Specifically for the SecureRandom class, it is known that a static seed will produce the same PRNG output [4, 8].

I5. Not processing the internal buffers after encryption or decryption. When Java's Cipher is used for cryptography, the proper call of the dofinal() function, which processes the last block in the buffer (i.e. ciphertext or plaintext), should not be omitted for both the encryption and the decryption phase. The internal mechanism of the algorithm implementation, depending on its encryption mode (ECB, CBC, or other), keeps an internal buffer which must also be discarded[6].

I6. Use of RSA with a padding other than OAEP. This should be avoided due to the fact that the use of a padding, such as PKCS1Padding, which does not use random bytes, will delay the adversary to decrypt the data or infer patterns from the ciphertext less than the OAEP padding will [11].

Weak keys. This category includes those cases where weak cryptographic keys are used, a practice that can put in risk the security of users and applications.

K1. Use of short keys. Yet another possible vulnerability of a cryptographic algorithm is short keys employment. According to the contemporary cryptographic standards [14], a key is weak when its length is less than 128 bits. The usage of a suchlike cryptographic key weakens the encryption and must be strictly avoided. For example, DES is known to have a set of weak keys, as it uses a 56-bits key, which does not provide sufficient security [8].

K2. Use of hard-coded encryption keys. The secrecy of encryption keys is an important factor and this practice can result even in the disclosure of the key to the adversary [8]. The encryption keys must be dynamically generated and developers should strictly avoid exposing them in the application's code [4, 9].

K3. The use of static/constant encryption keys. It is possible for an encryption key to be static without being hard-coded, e.g. when a byte array is initialized and remains the same for the whole process. The randomness of the encryption keys is the major factor contributing to encryption schemes security, thus cryptographic keys should not be constant [4, 9].

K4. The use of hard-coded passwords for PBE. Although PBE is usually based on a password given by the user as an input to the Android application, there are cases where developers use a specific value defined statically. In this way, developers make the application use the same password for each execution, while the password value can easily be accessed by the adversary.

Weak cryptographic parameters. This category comprises weaknesses related to poor choice of cryptographic parameters, like cryptographic modes, IVs, and seeds.

P1. Use of block ciphers with Java's default cryptographic mode. When only the cipher algorithm is invoked (without a specific mode defined), the default cryptographic mode used in specific providers (SunJCE and SunPKCS11) is the ECB[7,8], which is considered unsafe.

P2. Use of CBC encryption mode together with a non-random IV. An IV should be neither static nor predictable (for example an IV consisting of 0's or sequential numbers) [4, 9], otherwise the resulting cryptographic scheme is not considered safe.

P3. Use of CTR encryption mode together with a static counter value. It does not constitute a safe cryptographic scheme as it is not IND-CPA secure.

P4. Use of hard-coded IVs. Developers have to generate IVs dynamically for two reasons: (a) preventing adversaries from obtaining the specific primitive's value, and (b) generating different values for the IV in each cryptographic stage [4].

P5. Use of constant IV. A constant IV or an IV reuse renders many cryptographic schemes IND-CPA insecure, as the IV constitutes the only primitive introducing randomness in a cryptographic procedure and using a constant or a static IV frequently results in producing the same ciphertext. An IV

[6] http://www.itcsolutions.eu/2011/08/24/how-to-encrypt-decrypt-files-in-java-with-aes-in-cbc-mode-using-bouncy-castle-api-and-netbeans-or-eclipse/

[7] http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html

[8] http://docs.oracle.com/javase/8

can be constant without being hard-coded if, for example, is randomly generated but used more than once.

P6. Deriving IVs from keys or messages. This practice makes the IV non-random and predictable [4, 9] and is considered to be insecure.

P7. Generating IVs from cipher's blocksize, based on byte array creation. Many developers generate the IVs manually by initializing a vector having the size of cipher's blocksize with the default values of the creation of a byte array (bytearray = new byte[]), in combination with nextbytes() method of Random class. There are also cases where not even the Random class is utilized. It has to be noted that Random class use is not a proper practice, while deriving the IV without introducing any randomness, using Java default values to a byte array, makes the IV non-random and predictable [4, 9].

P8. Use of predictable PRNG seeds. The seed of the PRNG constitutes an important factor in constructing a secure cryptographic scheme. Developers should use non predictable seeds with PRNGs, so as to generate a high entropy key and not weaken PRNG's strength [8]. It is also essential to note that the setSeed() method of the SecureRandom Java class produces a predictable seed and must not be used in the key generation process [12].

5. METHODOLOGY & EXPERIMENTS

Our approach is organized in four main phases:

1. Application collection
2. Application utilization
3. Static analysis
4. Dynamic analysis

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

5.1 Application Collection

This initial phase of our study comprises the selection of the particular type of applications, as well as the applications themselves that were put under examination. We chose to analyze 49 Android applications of four different categories related to data encryption:

1. *Secure messaging:* This category includes applications that exchange encrypted data either via SMS, or through Bluetooth and Internet services (chat, social media and email). This category comprises 23 applications.
2. *Document encryption:* Document encryption describes applications that are involved with any kind of document encryption, like file encryption, directory encryption, multimedia content encryption, and note encryption. We downloaded 7 applications belonging to this category.
3. *Sensitive data exchange & storage:* Applications that appertain to this particular category are those handling any type of sensitive data (passwords, credit card numbers, pins etc.). 13 applications belong to this category.
4. *Multipurpose encryption utility:* This particular class contains applications offering more than one operations such as generating passwords, document encryption, text encryption, sensitive data storage, password vaults etc. This category comprises 5 applications.

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

5.2 Application Utilization

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

One of the checked parameters for all applications is the *utilization of a password.* Applications encompassing encryption usually utilize a password consisting of letters, digits, or alphanumeric characters. A password is introduced by the user and commonly takes part in the process of the plaintext encryption. There are many cases, however, where the password is only used as a pin.

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

In order to ascertain ciphertext's indistinguishability through each application's graphical environment, we considered three different scenarios:

1. input of the same plaintext twice in the application under examination, without disrupting its operation
2. input of the same plaintext once before and once after application and device reboot, and
3. execution of the application in two different Android devices, inserting in both cases the same plaintext.

5.3 Static Analysis

In the third phase of our study we proceeded in the source code auditing with the intention of further inspecting, in full detail, the cryptographic primitives in use (i.e. the general encryption scheme employed, the cryptographic algorithms, their parameters and specific modes of operation). Static analysis involves the following three steps:

1. **Obtaining the target application's .apk file.** We downloaded and installed the mobile application Root File Explorer in a rooted Android phone, so as to be able to explore the device's files and copy the target application's .apk from /data/app/ to the SD card.
2. **Extracting the source code of the application by apk decompilation.** For this step, we used dex2jar[9] toolset as

[9] https://github.com/pxb1988/dex2jar

well as JD-GUI tool from Java Decompiler[10] project. Specifically, dex2jar is a set of tools to convert Android .dex files into Java .class files, while JD-GUI is a standalone graphical utility that displays Java .class files source code. The step sequence followed in this stage for each application was the following (see Figure 1):

- Add the extension ".zip" to the .apk file (so that example.apk becomes example.apk.zip) and extract the zip file into folder *example_folder*.
- Copy the files of the toolset dex2jar into *example_folder*.
- From the command prompt, execute the command *dex2jar classes.dex*. This command will generate the file *clasess.dex.dex2jar* into the *example_folder*.
- Finally, we obtain access to Java source code by opening the *classes.dex.dex2jar* file using the JD-GUI application.

3. **Source code analysis and reviewing.** The last and most important step is the examination of the obtained source code of the applications, in order to evaluate their security based on the list of the cryptographic weaknesses discussed in the previous section.

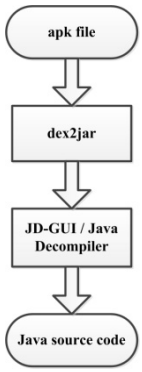

Figure 1. apk decompilation

5.4 Dynamic Analysis

In this phase we performed dynamic analysis in order to verify the obtained results from the static analysis or to examine cases where the results of the static analysis were inconclusive. That is, static analysis cannot always cover the whole functionality of the application. The reason behind this is that many Android applications make use of native code which is not available after the apk decompilation process that we followed in the previous phase. What is more, there is always the possibility to include in the source code functions that are not actually called during application's execution. With a view to include in our research these cases as well and have more accurate results, we combined static with dynamic analysis techniques.

As far as the dynamic analysis procedure is concerned, we chose to install and run the Android applications under examination on emulators, and monitor their functionality through Dalvik Debug Monitor Server (DDMS) of Android Software Development Kit (SDK). Particularly, via the Track Memory Allocation functionality we were able to detect the exact cryptographic algorithms and functions invoked. In addition, due to the fact that DDMS does not support native heap debugging anymore, we used androguard[11], a reverse engineering tool for Android, in order to detect native code usage.

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

6. RESULTS & EVALUATION

This section presents the results produced by the static and dynamic analyses we performed. Due to specific particularities of some applications, however, there were cases where applications were exclusively examined by employing static analysis (9 applications out of 49). The findings of our study are presented in Table 1, including those applications on which static analysis only was applied.

Table 1. Individual weaknesses per application category

Category / Weakness	SM	MEU	DE	SDES	Total
C1	17	6	7	2	**32**
C2	3		1		**4**
C3	9		4	3	**16**
C4	2				**2**
C5	8		3		**11**
C6	2		3	1	**6**
I1	3				**3**
I2	5	1	2		**8**
I3	2	1	3		**6**
I4	2				**2**
I5			1		**1**
I6	2				**2**
K1	2	2	2	1	**7**
K2	2		1	1	**4**
K3	1		2	1	**4**
K4	1		1		**2**
P1		2	4	1	**7**
P2	3				**3**
P3	2				**2**
P4	1		1		**2**
P5	1		1		**2**
P6					**0**
P7	1	1			**2**
P8	3	1			**4**

Legend:
SM: Secure messaging
MEU: Multipurpose encryption utilities
DE: Document encryption
SDES: Sensitive data exchange & storage

One of the first observations is that the most common weaknesses are: C1 (weak cryptographic algorithm or hash function) which is detected in 32 applications (65.3%), C3 (cryptographic algorithm in ECB mode) in 16 applications (32.7%), and C5 (CBC mode

[10] http://jd.benow.ca/

[11] https://github.com/androguard/androguard

with PKCS5Padding) in 11 applications (22.4%). Interestingly enough, these three weaknesses belong to the same category (weak cryptography). By grouping weaknesses into the categories presented in Section 4, it can be seen (Table 2) that most observed misuses in Android applications are related to weak cryptography, followed by weak implementations of the algorithms and weak cryptographic parameters selection; the least observed weaknesses are related to the selection of weak cryptographic keys.

Table 2. Grouped weaknesses per application category

Category / Weakness	SM	MEU	DE	SDES	Total
Weak crypto	41	6	18	6	**71**
Weak implementations	14	2	6	0	**22**
Weak keys	6	2	6	3	**17**
Weak parameters	11	4	6	1	**22**

The results of the application testing scenarios discussed in Section 5.2 are presented in Table 3 and concern those applications that are non IND-CPA secure based on their output. We deduced that the 30.6% of applications (i.e. 15 out of 49 applications) are not IND-CPA secure. From these non IND-CPA secure applications, 80% of them fail in all three scenarios, i.e. given the same plaintext as input, the same ciphertext is produced regardless if the user restarts the application or the device, or use another device, or not.

Table 3. Non IND-CPA secure applications

Scenarios not satisfied	No of apps
1, 2 and 3	12
1 and 2	1
1	2

The vast majority of Android applications' source code encompasses at least one cryptographic misuse, not always relevant to cryptographic algorithm and mode selection. Nonetheless, there are many cases where no cryptography is detected or out of date algorithms are invoked, for instance Caesar Cipher, Columnar Transposition, AtBash Cipher and Playfair Cipher along with others, even by applications bearing a name that implies the use of strong cryptography.

Although the majority of applications use AES in CBC mode, there is a significant number of Android applications that include either ECB mode or at least one obsolete algorithm. As far as a more comprehensive and statistical analysis of the results is concerned, it seems that the applications presenting a weakness related to cryptography misuse (i.e. all weaknesses apart C6) reach a percentage of 87.8% (i.e. 43 out of 49 applications). At the same time, the applications in which no cryptography was detected (i.e. weakness C6) reach the 12.2% (i.e. 6 out of 49 applications). Consequently, the percentage of the applications that seem to have no weakness is 0% (i.e. 0 out of 49 applications).

Another interesting point is that 95.9% of the tested applications (i.e. 47 out of 49 applications) present a weakness of the weak cryptography class (weaknesses C1 to C6). The percentage of applications that incorporate poorly implemented cryptography (weaknesses I1 to I6) is 32.7% (i.e. 16 out of 49 applications). Also, the 26.5% (i.e. 13 out of 49) of applications use weak cryptographic keys (weaknesses K1 to K4). Simultaneously, the percentage of applications employing cryptographic techniques with incorrect parameters (weaknesses P1 to P8) reach the 30.6% of the applications examined (i.e. 15 out of 49 applications).

Another interesting conclusion is the fact that although the most common cryptographic principle is that ECB mode of encryption is not IND-CPA secure and should not be used, the 32.7 % (i.e. 16 out of 49 application) make use of the specific mode in their cryptographic processes (weakness C3). An overview of all the aforementioned misuses is presented in Table 4.

Table 4. Cryptographic misuses findings overview

Misuse	Percentage of applications
Applications presenting at least one cryptography misuse weakness	87.8
Applications where no cryptography was detected	12.2
Applications where no weakness was detected	0
Weak or no cryptography usage detected	95.9
Weak implementations	32.7
Weak cryptographic keys usage	26.5
Incorrect cryptographic parameters employment	30.6
Use of ECB mode of encryption	32.7

7. COUNTERMEASURES

At this point, it is necessary to design a list of countermeasures and best practices that could be employed as a general methodology for developing Android applications using solid encryption. In the following we cite our proposals for specific cryptographic primitives' usage, emanating from our study:

1. Regarding **encryption algorithms**, developers should opt for AES and RSA for symmetric and asymmetric encryption respectively.
2. Depending on our previous selection, the most appropriate **encryption scheme** for AES is *CBC with PKCS7Padding*, while for RSA developers should select *OAEP padding*.
3. Another important practice that developers should certainly take into consideration is using **randomness** for any cryptographic parameter such as *passwords*, *encryption keys*, *initialization vectors*, *salts* and *seeds*. The aforementioned parameters must have the proper lengths and not be hard coded or statically defined in the source code so as not to use the same values for every execution of the application.
4. As for the random number generation, **Cryptographically Secure PseudoRandom Number Generators** (CSPNGs) should be used for encryption purposes.
5. As far as **Password-Based Encryption** (PBE) is concerned, the usage of proper parameters is required on behalf of the programmers. The password used for this particular procedure should not be hard coded, the iterations defined should be more than 1,000, and the salt should not be constant.
6. Last but not least, programmers should use only libraries that are known to use proper cryptographic techniques and follow all recommendations given by these libraries documentation (e.g. internal buffers processing after encryption or decryption).

The best practices presented in this section include essentially all types of cryptographic misuses observed in the applications examined, and summarize the entire set of rules in 6 principles

deemed adequate –if employed by the developers' community- to eliminate the high level percentages of cryptographic misuses.

8. CONCLUSIONS

In this paper, we have evaluated the use of cryptography in 49 Android applications whose operation is related to data encryption. The results showed that the majority of applications present at least one of those misuses. Developers' community lacks a specifically defined list of cryptographic misuses that must be avoided, as well as a list of best practices for cryptographic techniques. To this end, we provide guidelines, mainly intended for developers, to help them build more secure applications.

9. ACKNOWLEDGMENTS

This research has been partially funded by the European Commission in part of the SMART-NRG project (FP7-PEOPLE-2013-IAPP GA number 612294), the UINFC2 project (GA number HOME/2013/ISEC/AG/INT/4000005215), and the ReCRED project (Horizon H2020 Framework Programme of the European Union under GA number 653417).

10. REFERENCES

[1] Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G. and Tsay, J.-K. 2012. Efficient Padding Oracle Attacks on Cryptographic Hardware. *Advances in Cryptology – CRYPTO 2012*. R. Safavi-Naini and R. Canetti, eds. Springer Berlin Heidelberg. 608–625.

[2] Bellare, M., Desai, A., Pointcheval, D. and Rogaway, P. 1998. Relations among notions of security for public-key encryption schemes. *Advances in Cryptology — CRYPTO '98*. H. Krawczyk, ed. Springer Berlin Heidelberg. 26–45.

[3] Das, S., Gopal, V., King, K. and Venkatraman, A. 2014. *IV = 0 Security: Cryptographic Misuse of Libraries*. Technical Report #6.857 final project. MIT.

[4] Egele, M., Brumley, D., Fratantonio, Y. and Kruegel, C. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), 73–84.

[5] Hofheinz, D. and Unruh, D. 2008. Towards Key-Dependent Message Security in the Standard Model. *Advances in Cryptology – EUROCRYPT 2008*. N. Smart, ed. Springer Berlin Heidelberg. 108–126.

[6] John's Cryptography Blog: AES CBC Padding Oracle Attack: *http://johnx.blogspot.gr/2010/10/aes-cbc-padding-oracle.html*. Accessed: 2015-09-10.

[7] Klima, V. and Rosa, T. 2003. Side Channel Attacks on CBC Encrypted Messages in the PKCS#7 Format. *Cryptology ePrint Archive, Report 2003/098* (2003).

[8] Lazar, D., Chen, H., Wang, X. and Zeldovich, N. 2014. Why Does Cryptographic Software Fail?: A Case Study and Open Problems. *Proceedings of 5th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2014), 7:1–7:7.

[9] Li, Y., Zhang, Y., Li, J. and Gu, D. 2014. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. *Network and System Security*. M.H. Au, B. Carminati, and C.-C.J. Kuo, eds. Springer International Publishing. 349–362.

[10] McConnell, S. 2004. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press.

[11] MITRE - CWE-780: Use of RSA Algorithm without OAEP (2.8): *http://cwe.mitre.org/data/definitions/780.html*. Accessed: 2015-09-10.

[12] MOTOROLA 2012. *Best practices for encryption in Android*. White Paper.

[13] Nishika and Yadav, R.K. 2013. Cryptography on Android Message Applications – A Review. *International Journal on Computer Science and Engineering*. (2013), 362–367.

[14] NIST Cryptographic Standards and Guidelines Development Process: *http://www.nist.gov/director/vcat/cryptographic-standards-guidelines-process.cfm*. Accessed: 2015-09-09.

[15] Padding oracle attacks: in depth: *https://blog.skullsecurity.org/2013/padding-oracle-attacks-in-depth*. Accessed: 2015-09-10.

[16] Rizzo, J. and Duong, T. 2010. Practical Padding Oracle Attacks. *Proceedings of the 4th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2010), 1–8.

[17] Security Tips | Android Developers: *http://developer.android.com/training/articles/security-tips.html*. Accessed: 2015-09-09.

[18] Shuai, S., Guowei, D., Tao, G., Tianchang, Y. and Chenjie, S. 2014. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. *IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC)* (Aug. 2014), 75–80.

[19] The Padding Oracle Attack - why crypto is terrifying: *http://robertheaton.com/2013/07/29/padding-oracle-attack/*. Accessed: 2015-09-10.

[20] Vaudenay, S. 2002. Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS... *Advances in Cryptology — EUROCRYPT 2002*. L.R. Knudsen, ed. Springer Berlin Heidelberg. 534–545.

[21] Viega, J. 2003. Practical random number generation in software. *Computer Security Applications Conference, 2003. Proceedings. 19th Annual* (Dec. 2003), 129–140.