CRYPTOGRAPHIC MISUSE DETECTION TOOLS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EMİNE SELİN KORU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

SEPTEMBER 2021

Approval of the thesis:

CRYPTOGRAPHIC MISUSE DETECTION TOOLS

submitted by **EMİNE SELİN KORU** in partial fulfillment of the requirements for the degree of **Master of Science in Cryptography Department, Middle East Technical University** by,

Prof. Dr. Sevtap Selçuk Kestel
Director, Graduate School of **Applied Mathematics** _______________

Prof. Dr. Ferruh Özbudak
Head of Department, **Cryptography** _______________

Assoc. Prof. Dr. Murat Cenk
Supervisor, **Cryptography, METU** [illegible]

Dr. Pınar Gürkan Balıkçıoğlu
Co-supervisor, **Private Sector** [illegible]

Examining Committee Members:

Assoc. Prof. Dr. Oğuz Yayla
Institute of Applied Mathematics, METU [illegible]

Assoc. Prof. Dr. Murat Cenk
Institute of Applied Mathematics, METU [illegible]

Assoc. Prof. Dr. Fatih Sulak
Department of Mathematics, Atılım University F Sulak

Date: _______________

ABSTRACT

CRYPTOGRAPHIC MISUSE DETECTION TOOLS

Koru, Emine Selin
M.S., Department of Cryptography
Supervisor : Assoc. Prof. Dr. Murat Cenk
Co-Supervisor : Dr. Pınar Gürkan Balıkçıoğlu

September 2021, 54 pages

Today digital devices are an inevitable part of our lives. We use these devices for things like sharing photos, communicating with friends, and exchanging money. All these actions need privacy. In fact, properly used cryptographic systems can fulfill this need for privacy. However, in some cases developers can make mistakes, because of the reason such as lack of knowledge about cryptography and hard usage of cryptographic APIs. Thus, cryptographic misuse detection has become a new field of study and tools have begun to be designed to detect these misuses. In this study, we discuss 11 detection tools. They are CryptoLint [61], CMA [87], CDRep [77], sPECTRA [68], BinSight [79], CHIRON [85], CryptoGuard [84] and CRYLOGGER [83] in Android platform; iCryptoTracer [76] and Automated Binary Analysis [65] on iOS platform and CRYPTOREX [94] on IoT platform. Moreover, we give a comprehensive rule set and discuss shortcomings of each tool to contribute to future studies.

Keywords: reverse engineering, mobile security, Android applications, cryptographic misuse

ÖZ

KRİPTOGRAFİK YANLIŞ KULLANIM TESPİT EDEN ARAÇLAR

Koru, Emine Selin
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Murat Cenk
Ortak Tez Yöneticisi : Dr. Pınar Gürkan Balıkçıoğlu

Eylül 2021, 54 sayfa

Günümüzde dijital cihazlar hayatımızın kaçınılmaz bir parçasıdır. Bu cihazları fotoğraf paylaşmak, arkadaşlarımızla iletişim kurmak ve para alışverişi yapmak gibi işler için kullanırız. Tüm bu eylemlerin gizliliğe ihtiyacı vardır. Aslında, doğru şekilde kullanılan kriptografik sistemler bu gizlilik ihtiyacını karşılayabilir. Ancak bazı durumlarda geliştiriciler, kriptografik bilgi eksikliği ve kripto API'larının zor kullanımı gibi nedenlerle hata yapabilirler. Bu yüzden kriptografik yanlış kullanımların tespiti yeni bir çalışma alanı haline gelmiş ve bu yanlış kullanımları tespit etmek için araçlar geliştirilmeye başlanmıştır. Bu çalışmada, 11 kriptografik yanlış kullanım tespit aracı üzerine çalışılmıştır. Bahsedilen bu 11 araç, Android platformunda CryptoLint [61], CMA [87], CDRep [77], sPECTRA [68], BinSight [79], CHIRON [85], CryptoGuard [84] ve CRYLOGGER [83]; iOS platformunda iCryptoTracer [76] ve Automated Binary Analysis [65] ve IoT platformunda ise CRYPTOREX [94]'tir. Ayrıca, çalışmamızda kapsamlı bir kural seti tanımlanmış ve gelecekteki çalışmalara katkı sağlamak amacıyla her bir aracın eksiklikleri de tartışılmıştır.

Anahtar Kelimeler: tersine mühendislik, mobil güvenlik, Android uygulamaları, kriptografik yanlış kullanım

To my family, especially to my dear husband

ACKNOWLEDGMENTS

I would like to express my very great appreciation to my thesis supervisor Assoc. Prof. Dr. Murat Cenk and and co-supervisor Dr. Pınar Gürkan Balıkçıoğlu for their patient guidance, enthusiastic encouragement and valuable advices during the development and preparation of this thesis. Their willingness to give them time and to share their experiences has brightened my path.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
ECB	Electronic Code Book
PBE	Password-Based Encryption
PRNG	Pseudo Random Number Generator
CBC	Cipher Block Chaining
IV	Initialization Vector
MD	Message Digest
SHA	Sevure Hash Algorithm
RSA	Rivest–Shamir–Adleman
PKCS	Public Key Cryptography Standards
OAEP	Optimal Asymmetric Encryption Padding
NIST	National Institute of Standards and Technology
MITM	Man In The Middle
HTTP	Hyper Transfer Protocol
SSL	Secure Socket Layer
TLS	Transport Layer Security
RC2	Ron's Code 2
DES	Data Encryption Standard
JCA	Java Cryptography Architecture
CFG	Control Flow Graph
sCFG	Super Control Flow Graph
CMA	Crypto Misuse Analyzer
APK	Android Application Package
TBC	Target Branch Collection
UI	User Interface
ARM	Advanced RISC Machines
LLVM	Low Level Virtual Machine
IR	Intermediate Representation

ICFG	Inter-Procedural Control Flow Graph
SEAM	Symmetric Encryption Algorithms Misuses
HAM	Hash Algorithms Misuses
AEEM	Asymmetric Encryption Algorithms Misuses
PKMM	Password and Key Management Misuses
PBEM	Password Based Encryption Misuses
PRNGM	Pseudo-random Number Generator Misuses
ICM	Internet Connection Misuses
OM	Other Misuses

CHAPTER 1

INTRODUCTION

Sharing confidential data is one of the most important issues in human history. With the advancement of technology, digital devices have started to take place in data sharing very frequently. For example, electronic banking transactions can be performed through an application installed on the device or passwords can be stored with another application. In this and all similar processes, application developers resort to cryptographic algorithms and APIs to protect confidential data since they provide data privacy and data integrity between two entities. However, misuse of cryptographic endangers the whole security of the system.

In 2014, Lazar et al. [73] analyzed 269 cryptographic misuse and they revealed that 223 of them happen because of misuse of a security library by developers and just the rest, i.e only 46, are caused by faulty library implementations. According to Egele et al. [61], 88% of the Android applications in the 11.748 that were downloaded from Google Play Store [14] and used cryptographic APIs had at least one misuse. Veracode [47] showed that cryptographic misuses are the second most common vulnerability. Another study [83] analyzed 15.4 % applications in the 1.3 million Android applications which use security-related code snippets taken from Stack Overflow [37] and found that 97.9 % have at least one misuse. Moreover, according to Nadi et al. [80], 65% developers found cryptographic APIs hard to use and when asked how to simplify API usage, developers indicated that tools which can automatically detect and fix misuse can make the usage of the cryptographic API easy.

As a result of these, there are lots of tools developed and continue to be developed to detect cryptographic misuses. The idea behind the tools is defining cryptographic rules which use to check whether an application complies with these rules or not. Thus, to fulfill this duty, static and dynamic tools are developed. Due to the nature of this work, both of the static and dynamic tools have shortage and surplus.

Static analysis tools can analyze code without executing it and for this reason they can analyze a large number of programs according to the wide range of security rules. However, a static approach can cause false positives which means it can flag a cryptographic algorithm is misused even if it is not. Moreover, it can miss some cryptographic misuses which are loaded dynamically. Although the negative sides of static analysis, it is commonly used by developers since it gives better perception of code structure. By using static analysis developers can locate weaknesses exactly in the application and find all execution paths.

On the other hand, dynamic analysis needs to be executed and has to be triggered at runtime to detect existing cryptographic misuses. For this reason it cannot be analyzed with a large number of programs with respect to the wide range of rules. However, it generates a few false positives. Moreover, the response time, system memory and the behaviour of a function can be easily seen. Also if there is no source code of an application, it can still be analyzed and it can be used to validate the result of static approaches.

In this study, the rule sets and detailed methodologies used to detect cryptographic misuses and the most effective cryptographic misuse detection tools are examined. The tools and associated analysis methods covered in this study are shown in Table 1.1. Android-Java, iOS and IoT are selected as working platforms. For the Android-Java platforms CryptoLint [61], CMA [87], CDRep [77], sPECTRA [68], BinSight [79], CHIRON [85], CryptoGuard [84] and CRYLOGGER [83] are the detection tools that is studied, for the iOS platform iCryptoTracer [76] and Automated Binary Analysis [65] (for ease of use it is called as Framework in this study) are selected and lastly for the IoT part CRYPTOREX [94] is selected. Selected tools are mentioned chronologically on a platform basis.

Table 1.1: Cryptograptic misuse detection tools for Android, iOS, and IoT (respectively yellow, pink, and blue) applications

	Static	Dynamic	Hybrid
CryptoLint [61]	✓		
CMA [87]			✓
CDRep [77]	✓		
sPECTRA [68]			✓
BinSight [79]	✓		
CHIRON [85]	✓		
CryptoGuard [84]	✓		
CRYLOGGER [83]		✓	
iCryptoTracer [76]			✓
Framework [65]	✓		
CRYPTOREX [94]	✓		

1.1 Contributions

The main contributions of this work are following:

- To the best of our knowledge this study is the first work that collects all the 11 tools that have been released so far for Android-Java, iOS and IoT platforms. The aim of this study to give the capsule information that developers of detection tools need. In this study, it is revealed which rules are checked by mentioned tools to detect cryptographic misuse, and which methodology is used to detect violations of these rules.

- We create a comprehensive set of cryptographic misuses. In this way, it can be easily determined which rules should be added to the future works that are not available in the tools so far.
- We have revealed the shortcomings of each tool to guide future studies. In the light of this information, this study can be a source for the features and methodology of a new tool to be developed.
- Using this study, developers can decide which methods and tools they will use to determine whether their applications have cryptographic vulnerabilities or not.

1.2 Overview

This thesis is organized as follows. In Chapter 2, the defining comprehensive rule set which is used by the tools is showed. Then it is split into three parts according to the belonging platform of each tool. In Chapter 3, the tools which can detect cryptographic misuse in Android-Java platforms are showed, in Chapter 4 tools work on iOS platform and lastly in Chapter 5 there is just one tool that works on the IoT platform. As a contribution, shortcoming of tools in Chapter 6 and related works in Chapter 7. Finally, Chapter 8 is the conclusion part.

CHAPTER 2

RULES OF TOOLS

Cryptographic misuses are defined as the improper usage of cryptographic algorithms. Developers use cryptographic APIs or libraries in a predefined way to apply properly. Although cryptographic algorithms and their APIs are well implemented and well-defined, the lack of knowledge about usage of cryptographic API and cryptographic algorithms make developers prone to misuse.

In this section, we define the rules that have to be obeyed to prevent cryptographic misuse. These rules mentioned in this section will be used by the tools in Chapter 3, Chapter 4 and Chapter 5 to detect cryptographic misuse.

The rules used in the tools checking misuse mechanism have been tried to be showed in this section comprehensively. All other rules have emerged from these rules with minor additions. If necessary some of the rules which differ from the below rules will be explained with more detail in the rules section of the corresponding tools. In total, there are 33 rules and Figure 2.2 has been created to better understand which tool uses which rule. For ease of use each rule called as "R-number of the rules" in the table. For example, the first rule (Don't use ECB mode for encryption) is called as R-01.

Cryptographic misuses are divided into 8 types. They are Symmetric Encryption Algorithms Misuses (SEAM), Hash Algorithms Misuses (HAM), Asymmetric Encryption Algorithms Misuses (AEEM), Password and Key Management Misuses (PKMM), Password Based Encryption Misuses (PBEM), Pseudo-random Number Generator Misuses (PRNGM), Internet Connection Misuses (ICM) and finally Other Misuses (OM). The distribution of the misuses according to the defined types are in Figure 2.1. The Password and Key Management Misuses is the most common misuse as in shown in the Figure 2.1. Moreover, detailed information about current weaknesses, especially cryptography, can be found in CWE Version 4.5 [12].

1. **Symmetric Encryption Algorithms Misuses(SEAM)**

   R-01 Don't use ECB mode for encryption [7]: Even though ECB mode is widely used, it is not secure since the same plaintexts encrypt to the same ciphertexts and the blocks of plaintext which are encrypted correspond to the same block of ciphertext.

   R-02 Don't use the operation mode CBC(client/server scenarios) [89]: CBC can be exploited with the padding oracle attacks in client/server scenarios.

R-03 Don't use non-random IV for CBC mode [50, 60]: Using non-random IV in CBC mode make system vulnerable against dictionary attacks.

2. **Hash Algorithms Misuses (HAM)**

   R-04 Don't use broken hash functions (SHA1, MD2, MD5, etc.) [29, 83]: For example, MD4 is not one-way and MD5 has many collisions and it can be attacked [11, 74, 90]. Therefore, it is not secure to use them.

3. **Asymmetric Encryption Algorithms Misuses(AEAM)**

   R-05 Don't use a short key ($<$ 2048 bits) for RSA [21, 53]: The key length of the RSA algorithm has to be more than 2048 bits.

   R-06 Don't use the textbook (raw) algorithm for RSA [56]: A padding algorithm has to be different from NOPADDING [60].

   R-07 Don't use the padding PKCS1-v1.5 for RSA [55]: PKCS1-v1.5 [69] can be used only for encryption and decryption.

   R-08 Don't use RSA algorithm without OAEP [46, 54, 58]: Usage of RSA with no Optimal Asymmetric Encryption Padding is insecure.

   R-09 Don't use digital signing algorithm SHA1withRSA [53]: The use of SHA-1 in digital signature creation is not permitted by NIST, as SHA-1 can be exploited by a collision attack.

4. **Password and Key Management Misuses(PKMM)**

   R-10 Don't use a static (constant) key for encryption [44]: In cryptographic systems security depends on the key. Using constant keys breaks the security of the system since an attacker can recover the encrypted data and also the key.

   R-11 Don't use a "badly-derived" key for encryption [33]: In symmetric encryption systems the keys have to be generated randomly if not then it can be said that the key derived badly.

   R-12 Don't reuse the initialization vector (IV) and key pairs [32, 28, 27]: Reusing the pair of (key, IV) makes the ciphertext predictable since the pair has to be changed for each message.

   R-13 Don't use a weak password [92, 48]: A password which is hard for PBE and is not in the black-listed is required.

   R-14 Don't use a NIST-black-listed password [26, 30]: There is a list for common password validation of NIST. The usage of these passwords is insecure.

   R-15 Don't reuse a password multiple times [48]: Passwords responsible to protect the system so reusing the password makes the system vulnerable.

   R-16 Don't use a static (constant) password for store [38]: For the key storage it is not allowed to use a static password since it can be recovered.

R-17 Don't use keys with insufficient key length [33]: If the key length is less than 128 bits, it increases the possibility of brute force attack.

R-18 Don't exchange keys without entity authentication [22]: Key exchange maintains confidentiality between client and server, but it cannot guarantee that they are who they claim to be. This situation makes key exchange prone to MITM(Man in the Middle) attack. That is, the victim trusts the malicious server as a trusted server. Moreover, the malicious server can request the authentication information of the victim and by using this information communicate with the server like the victim.

R-19 Don't use a key past its expiration date [40]: Some of the software use a password or cryptographic key which past its expiration date. It is insecure usage since it gives more time to the attacker to break security.

5. **Password Based Encryption Misuses(PBEM)**

   R-20 Don't use a static (constant) salt for PBE [45, 42]: In password based encryption the password concatenated with a salt. Using constant salt is the same as using no salt.

   R-21 Don't use a short salt (< 64 bits) for PBE [78, 42, 41]: To protect password salt has to be large enough.

   R-22 Do not use fewer than 1,000 iterations for PBE [45, 78]: In password based encryption the password with salt is subject to iterations of a cryptographic hash function. Using fewer than 1000 iterations makes password guessing attacks possible.

6. **Pseudo-random Number Generator Misuses (PRNGM)**

   R-23 Don't use a static (constant) seed for SecureRandom [86]: SecureRandom is a strong pseudo-random number generator class and designed to produce non-deterministic output. However, if the seed is constant, then output of the PRNG is also constant. That is, if PRNG is used to derive keys with a constant seed, the key would not be random in this case.

   R-24 Don't use an unsafe PRNG (java.util.Random) [86, 43]: `java.util.Random` class cannot be used to generate random numbers in cryptographic operations.

7. **Internet Connection Misuses(ICM)**

   R-25 Don't use HTTP URL connections (use HTTPS) [64]: HTTP is not a secure connection anymore, HTTPS has to be used instead of HTTPS.

   R-26 Don't verify host names in SSL/TLS in trivial ways [64]: It is necessary to verify hostnames. Accepting all hostnames is not allowed. By using MITM attack an attacker can intercept and modify network traffic between client and server.

   R-27 Don't verify certificates in SSL/TLS in trivial ways [64]: It is necessary to verify certificates. Accepting all certificates is not allowed. If certificates do not validate properly, an attacker can make an MITM attack. So, an attacker can intercept and modify network traffic between client and server.

R-28 Don't manually change the hostname verifier [64]: Modifying the standard hostname verifier is not allowed since it can cause insecure communication over SS-L/TLS.

8. **Other Misuses(OM)**

R-29 Don't use a static (constant) initialization vector [27]: In the symmetric systems the IV is used instead of keys. For example, In CBC mode each block of plaintext is XORed with the previous block of ciphertext. But, the first block of plaintext is XORed with the IV. Therefore, constant IV usage is not secure since it makes the result deterministic.

R-30 Don't use a "badly-derived" initialization vector (IV) [13]: Since IV is used as a key in a symmetric system, it has to be generated unpredictable sufficiently.

R-31 Don't use the same salt for different purposes [32]: Since it used to protect passwords by adding randomness to it, usage of the same salt is not allowed.

R-32 Don't use broken encryption algorithms(RC2 [70], RC4 [51], DES [70], etc.) [39]: Since the computing power increases, some algorithms can be attacked.

R-33 Vulnerable On-device Storage [68]: Private data of an application has not to be stored in external storage in Android since it is shared.

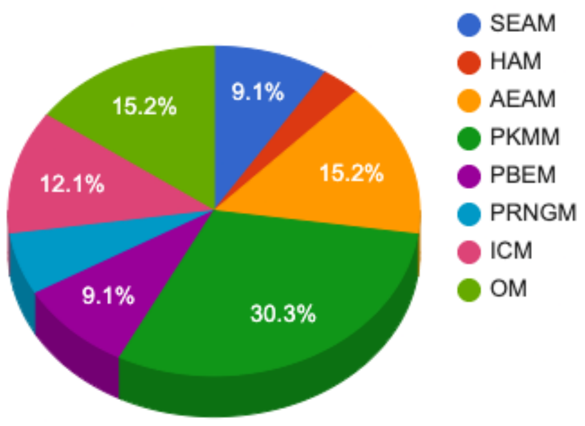
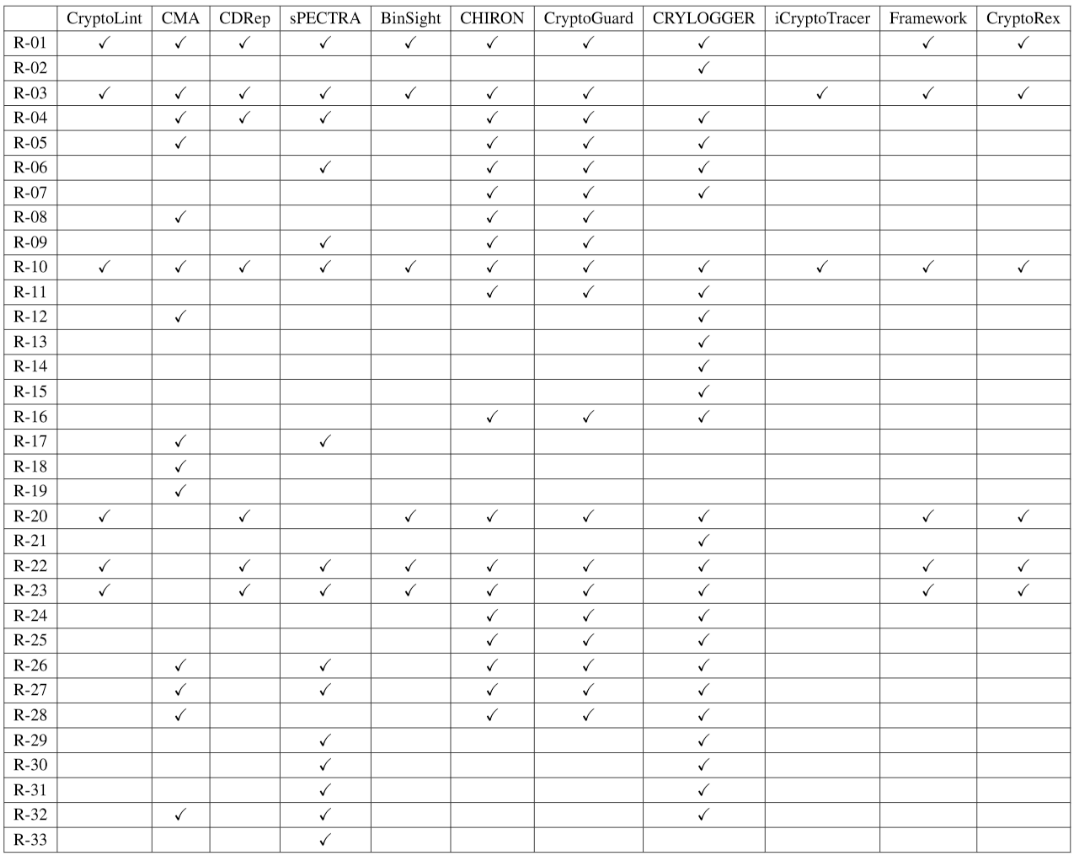


Figure 2.1: Distribution of rule categories



	CryptoLint	CMA	CDRep	sPECTRA	BinSight	CHIRON	CryptoGuard	CRYLOGGER	iCryptoTracer	Framework	CryptoRex
R-01	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
R-02								✓			
R-03	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
R-04		✓	✓	✓		✓	✓	✓			
R-05		✓				✓	✓	✓			
R-06				✓		✓	✓	✓			
R-07						✓	✓	✓			
R-08		✓				✓	✓				
R-09				✓		✓	✓				
R-10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R-11						✓	✓	✓			
R-12		✓						✓			
R-13								✓			
R-14								✓			
R-15								✓			
R-16						✓	✓	✓			
R-17		✓		✓							
R-18		✓									
R-19		✓									
R-20	✓		✓		✓	✓	✓	✓		✓	✓
R-21								✓			
R-22	✓		✓	✓	✓	✓	✓	✓		✓	✓
R-23	✓		✓	✓	✓	✓	✓	✓		✓	✓
R-24						✓	✓	✓			
R-25						✓	✓	✓			
R-26		✓		✓		✓	✓	✓			
R-27		✓		✓		✓	✓	✓			
R-28		✓				✓	✓	✓			
R-29				✓				✓			
R-30				✓				✓			
R-31				✓				✓			
R-32		✓		✓				✓			
R-33				✓							

Figure 2.2: Rules

CHAPTER 3

DETECTION TOOLS FOR ANDROID AND JAVA APPLICATIONS

Cryptographic misuse detection tools are mostly developed for Android and Java applications since they have a byte code format. Android applications are written as Java source code and it is compiled to Dalvik byte code. When a user installs an application from Google Play Store, an apk file is downloaded and installed in a device. The apk file is the packaged form of the Dalvik bytecode with the extra information, configuration files, images and some third party libraries.

Java Cryptography Architecture(JCA) identifies the usage of the cryptographic algorithms for developers. It points out how to invoke a cryptographic API on Android platform and ensures cryptographic services. These APIs can be used for encryption, key management or validation of a certificate. JCA implementations are included in the `java/security`, `javax/crypto` and `javax/security` package.

3.1 CryptoLint

CryptoLint is developed for detecting cryptographic misuses in Android applications. It is introduced by Egele et al. [61] and it is based on the Androguard Android program analysis framework [1]. Static program slicing is used for identifying flows between cryptographic operations, keys, initialization vectors and the other cryptographic materials. CryptoLint disassembles raw Android binary and checks whether cryptographic vulnerabilities exist or not. But, Android application can use native code also and CryptoLint cannot analyze native code.

For evaluating CryptoLint 145,095 applications from Google Play Store [14] are downloaded. Since the security rules of CryptoLint are related with the functionality of `javax/crypto` and `java/security` namespaces, the tool checks whether an application uses the functionality of these namespaces and eliminates unrelated apps. After these elimination 11,748 Android applications can be analyzed and 10,327 applications out of 11,748 have cryptographic vulnerabilities.

3.1.1 Rules

The rules which CryptoLint uses are as follows. Details of the rules are explained in Chapter 2, according to the rule number defined in the parenthesis.

1. Do not use ECB(Electronic Code Book Mode) mode for encryption (R-01)
2. Do not use a non-random IV for CBC (Cipher Block Chaining) encryption (R-03)
3. Do not use constant encryption keys (R-10)
4. Do not use constant salts for PBE (Password Based Encryption) (R-20)
5. Do not use fewer than 1,000 iterations for PBE (R-22)
6. Do not use static seeds to seed `SecureRandom` [31] (R-23)

3.1.2 Methodology

CryptoLint uses static program slices which terminate in calls to cryptographic APIs. Before explaining static program slices, Control Flow Graph (CFG) and super Control Flow Graph (sCFG) will be defined. CFG is a graphical representation. All the paths in the representation can be traversed in a program during its execution. Some types of CFG examples are shown in Figure 3.1

CryptoLint extracts intra-procedural CFG for all methods in an application. But, some cryptographic functionality might not be limited to a single method. For instance, the cipher object is used for both encrypt and decrypt methods. For this reason analyzing methods in an isolated way can not extract encryption schemes. Thus sCFG is used. sCFG is also a graphical representation but, it adds inter-procedural aspects to intra-procedural CFG.

Now, static program slicing can be expressed. According to the Weiser [91], in a program P, all statements which might affect the value of variable v in a statement x are included in the static program slice S. Slicing criterion is shown $C = (x, v)$. Static slices are calculated by back warding dependencies between statements. To compute a static slice for (x, v), it is necessary to investigate all statements which affect the value of v before statement x. Iteratively, for each statement y that can affect the value of v in statement x, it is necessary to compute the slices for each variable z in y which affect the value of v. Therefore, the static slice for (x, v) is union of all these slices.

3.2 Crypto Misuse Analyzer(CMA)

CMA is introduced by Shuai et al. [87] for performing analysis on Android applications. After selecting branches which invoke the cryptographic API, CMA runs the application using

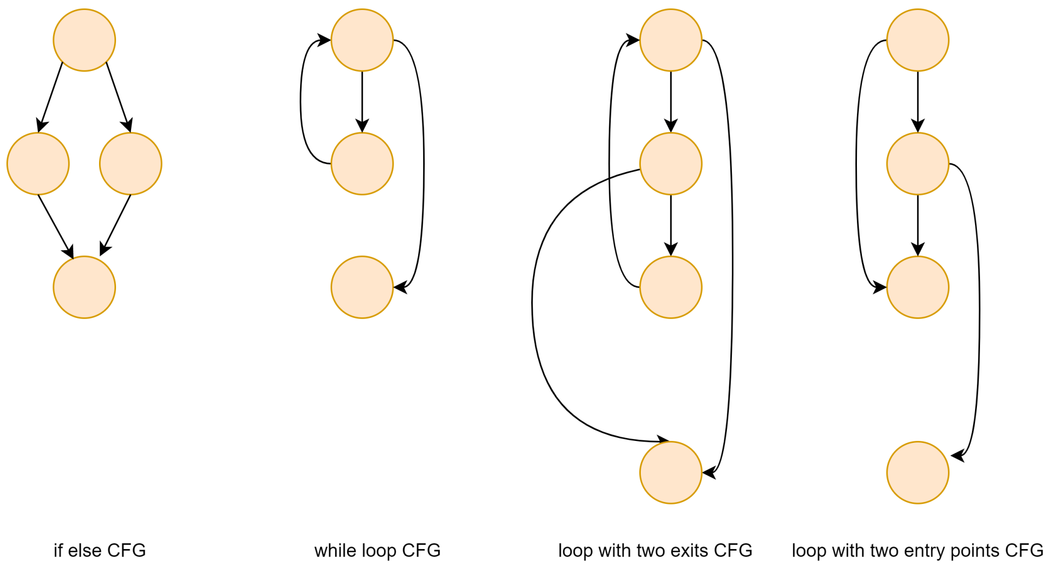


Figure 3.1: CFG examples

the target branch and records the cryptographic API calls. As a result of this, it determines cryptographic misuse by using records.

CMA performs a combination of static and dynamic analysis and it analyzes apk files directly.

3.2.1 Rules

The rules used in CMA are as follows. Details of the rules are explained in Chapter 2, according to the rule number defined in the parenthesis.

1. Use a non-random IV for CBC encryption (R-03)
2. Use ECB-mode for encryption (R-01)
3. Insufficient key (R-17)
4. Use a risk or broken algorithm for Symmetric encryption (R-32)
5. Use the same cryptographic key multiple times or hard-coded cryptographic keys for encryption (R-10)
6. Reversible one-way hash (R-04)
7. Inadequate key length (R-05)
8. RSA algorithm without OAEP(Optimal Asymmetric Encryption Padding) (R-08)
9. Improper Certificate Validation [16] (R-26, R-27 and R-28)
    - Missing Validation of Certificate [16]

- Improper Check for Certificate Revocation [17]
- Improper Validation of Certificate Expiration [19]
- Improper Validation of Certificate with Host Mismatch [20]
- Improper Following of a Certificate's Chain of Trust [18]

10. Use of Hard-coded Cryptographic Key or Password [44](R-10)
11. Key Exchange without Entity Authentication (R-18)
12. Reusing a Nonce, Key Pair in Encryption (R-12)
13. Use of a Key past its Expiration Date (R-19)

3.2.2 Methodology

CMA analyzes the symmetric encryption, hash and asymmetric encryption misuses. Firstly, CMA performs static analysis on an application, and checks whether cryptographic API is used or not.

If a cryptographic API is used, CMA builds Control Flow Graph(CFG) and Call Graph(CG), and it determines which branch has a cryptographic APIs. Afterwards, it runs the application through the detected branch, and generates runtime logs. Then, by using the rules defined in Chapter 3.2.1, CMA specifies whether there is a cryptographic misuse or not.

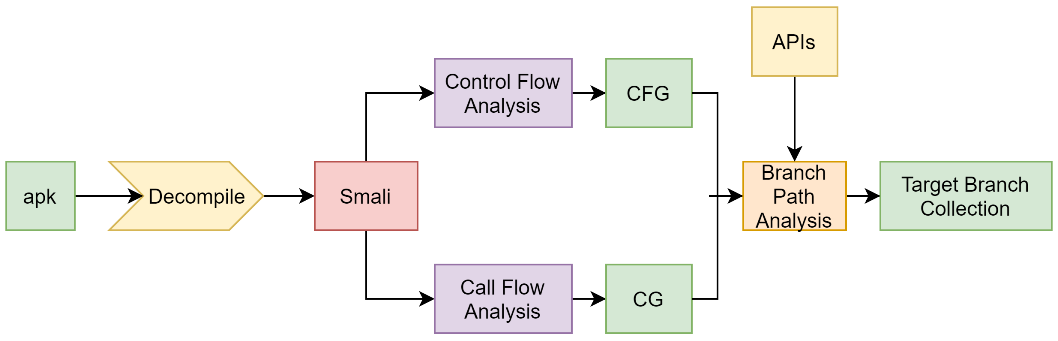


Figure 3.2: Brach Analysis

As shown in Figure 3.2, APK files are decoded by using apktool [6, 5] for achieving dalvik bytecodes. Then CMA can find the applications which invoke the cryptographic APIs. After selecting related applications, CMA constructs CFG and CG.

As mentioned in the previous tool, CFG is a graph representation. To define again, every Android application can be divided into small blocks such that each block contains dalvik instruction in a certain order and CFG shows relation between blocks. To construct CFG, jump instructions in an application can be taken into consideration since the destination address of a jump instruction can be observed statically.

CG is also a CFG, it shows calling relationships between subroutines of a program. CG consists of nodes and edges. Nodes are the entry point of a function. The edges are between two nodes. Besides that each edge records an entry point and exit point of a function, it also describes the relation between two nodes. CMA performs Branch Path analysis by using reverse traversal on CFG and CG for identifying which branch contains cryptographic API. At the end of path analysis, the Target Branch Collection(TBC) is constructed.

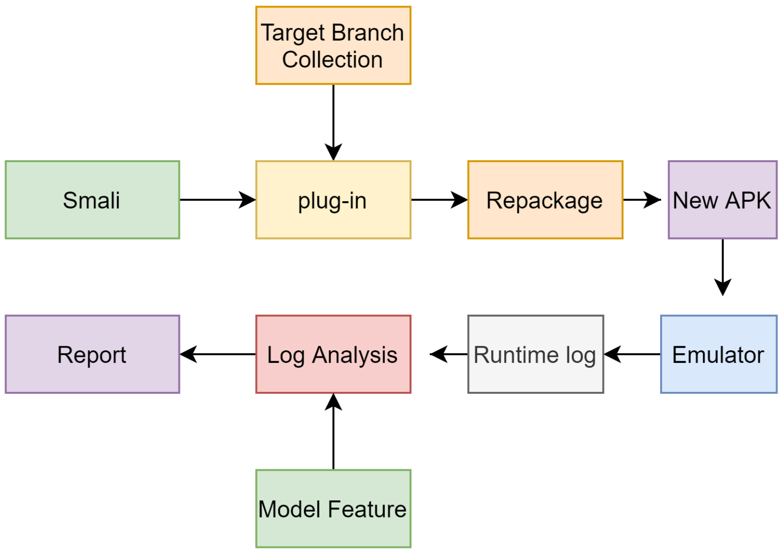


Figure 3.3: Process of CMA

The process of CMA is shown in Figure 3.3. It starts with adding logging codes into smali code by using a dynamic analysis tool, API Monitor [4]. After adding logging code, the application is repackaged and running on an emulator. In this step, the tool records runtime logs and compares it with the rules defined in the Chapter 3.2.1.

3.3 CDRep

CDRep is proposed by Ma et.al [77] and it is a misuse detection tool for Android applications but this tool differs from its ancestor because of the capability of repairing misuses after detecting it. It targets the byte code of the application rather than the source code. The procedure of CDRep is illustrated in Figure 3.4.

Research team of the CDRep analyzed 8640 Android applications, and they found that 8582 of them have cryptographic misuse. 1262 applications of 8582 applications are randomly choosed and manually checked for verification of CDRep repair capability. This manual check shows that 1193 applications are fixed correctly.

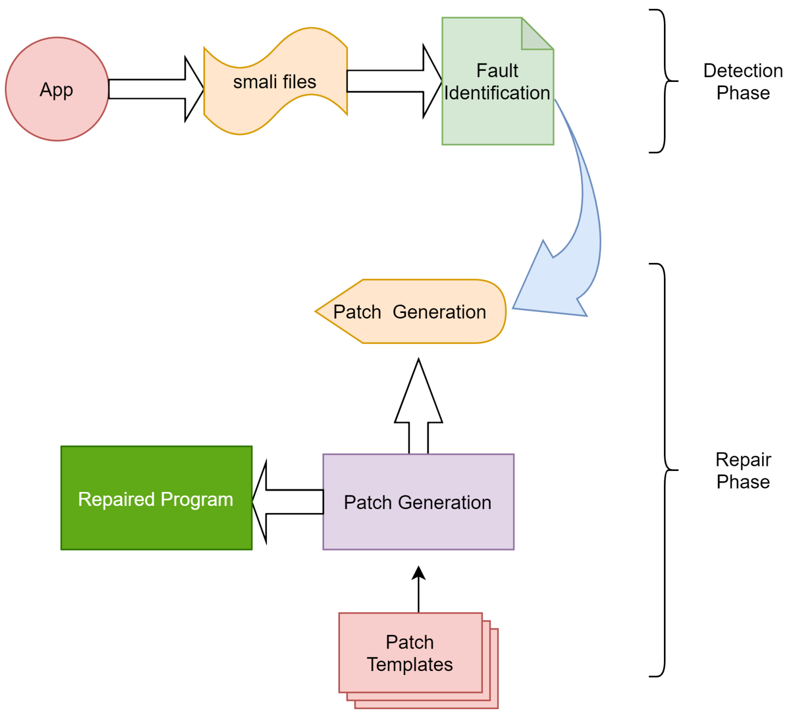


Figure 3.4: Process of CDRep

3.3.1 Rules

Details of the rules are explained in Chapter 2, according to the rule number defined in the parenthesis.

1. Do not use electronic code book (ECB) mode for encryption (R-01)
2. Do not use a constant Initialized Vector (IV) for ciphertext block chaining (CBC) encryption (R-03)
3. Do not use constant secret keys (R-10)
4. Do not use constant salts for password-based encryption (PBE) (R-20)
5. Do not use fewer than 1,000 iterations for PBE (R-22)
6. Do not use static seeds to seed `SecureRandom` (R-23)
7. Do not use reversible one-way hash (i.e. MD5, message-digest algorithm) (R-04)

3.3.2 Methodology

CDRep is based on 7 rules for misuse detection and it has 2 phases. First phase is the misuse detection phase. This tool can perform static analysis to locate the misuse and using the 7 rules, determine the type of the misuse. Second phase is the repairing phase. According to the best practices in cryptographic implementations, fix patterns are determined.

- **Detection Phase:**

  Before starting the detection phase, it is necessary to mention some terms. To detect the cryptographic misuse in a byte code, Indicator Instruction is needed. Indicator Instruction is the instruction related to the misuse. Root Cause Instruction is the instruction which causes the misuse.

  CDRep contains decompilation and fault identification steps and its procedure nearly the same as CryptoLint. CDRep checks whether vulnerabilities exist in the application or not, by decompiling the application. If there exists vulnerabilities, CDRep detects vulnerability type and Java class where vulnerability is in.

  Firstly, cryptographic vulnerabilities are detected in the decompiled code by locating indicator instructions. Then, other indicator instructions are detected for each such instruction. Until root causes are found, it is necessary to check all indicator instructions.

- **Repairing Phase:**

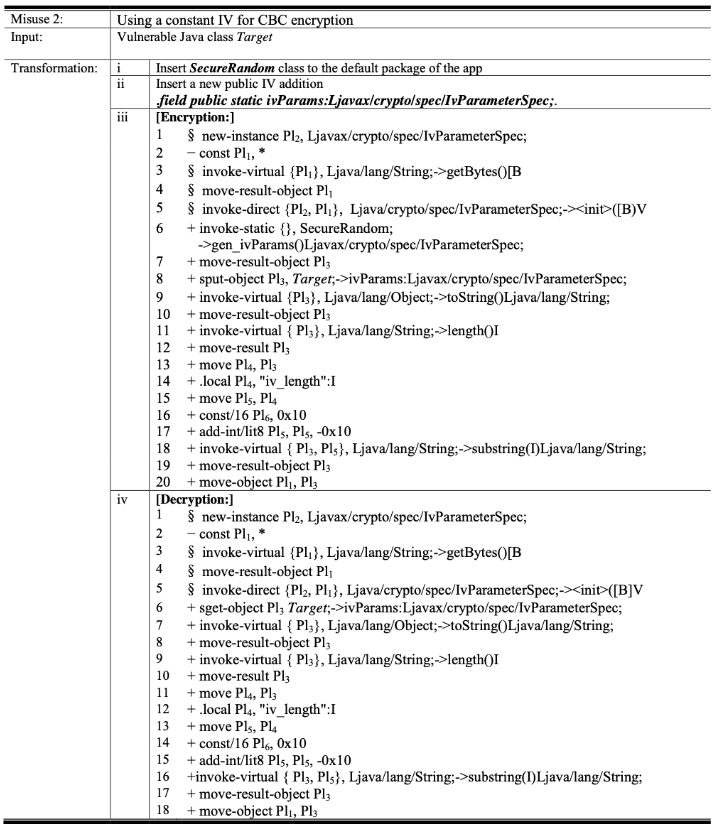
  There are 7 types of misuses CDRep can detect. Therefore, developers of CDRep produce 7 patch templates as shown in Table 3.1.

Table 3.1: Patch Templates [77]

Misuse	Patch
1	Use CTR mode
2	Random IV for CBC mode
3	Random secret key
4	Random salt for PBE
5	Iteration number more than 1000
6	Call $SecureRandom.nextBytes()$
7	Using SHA-256

They analyze the code of a target application and decide what to add and remove to turn the application into a well-written application. Generic patch is the generalization of the added and removed code and it contains transformations. In the generic patch register names replace with placeholders and constant values replace with a wildcard character ("*"). According to the article of CDRep [77], patch template for constant IV usage in CBC mode shown in Figure 3.5. It contains 4 transformations i, ii, iii, iv. Transformation i means that adding byte code of `java.security.SecureRandom` class to the application. Transformation ii means that adding field `IvParameterSpec` to vulnerable Java class. Transformation iii and iv imply that code addition (shown as +) and code deletions(shown as -). The lines of code shown by - replaced with the lines of codes shown by + . $Pl_1, ..., Pl_6$ in transformation iii are placeholders and "*" is the wildcard character in line 1. The root cause of misuse is in line 1 so, it is changed with line 6-20 for generating random value and stored in `ivParams` in line 8. Length of the randomized value is another topic to take into attention. Therefore, in line 11, the length of the randomized value is checked. If length is longer than required length, sub-length of randomized value is used as shown in line 15-18 and sub-length of randomized value will be IV. Transformation iv is the same procedure of transformation iii but, this time it is for decryption.

As shown in Figure 3.6, the parameters of transformation iii in the encryption step in Figure 3.5 are replaced with placeholders and wildcard character (Figure 3.6.A, Figure 3.6.B). Then, placeholders and wildcard characters are replaced with the actual register again and placeholders which do not represent actual registers replaced with other available registers like v_1, v_2 shown in Figure 3.6.C.

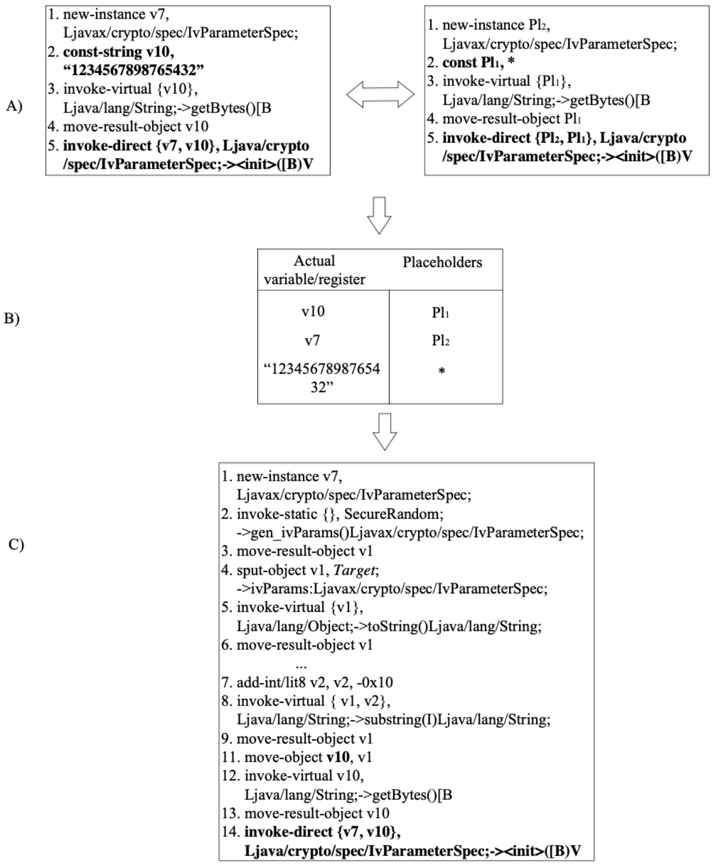


Misuse 2:	Using a constant IV for CBC encryption	
Input:	Vulnerable Java class *Target*	
Transformation:	i	Insert ***SecureRandom*** class to the default package of the app
	ii	Insert a new public IV addition ***.field public static ivParams:Ljavax/crypto/spec/IvParameterSpec;.***
	iii	**[Encryption:]** 1 § new-instance Pl$_2$, Ljavax/crypto/spec/IvParameterSpec; 2 − const Pl$_1$, * 3 § invoke-virtual {Pl$_1$}, Ljava/lang/String;->getBytes()[B 4 § move-result-object Pl$_1$ 5 § invoke-direct {Pl$_2$, Pl$_1$}, Ljava/crypto/spec/IvParameterSpec;-><init>([B)V 6 + invoke-static {}, SecureRandom; ->gen_ivParams()Ljavax/crypto/spec/IvParameterSpec; 7 + move-result-object Pl$_3$ 8 + sput-object Pl$_3$, *Target*;->ivParams:Ljavax/crypto/spec/IvParameterSpec; 9 + invoke-virtual {Pl$_3$}, Ljava/lang/Object;->toString()Ljava/lang/String; 10 + move-result-object Pl$_3$ 11 + invoke-virtual { Pl$_3$}, Ljava/lang/String;->length()I 12 + move-result Pl$_3$ 13 + move Pl$_4$, Pl$_3$ 14 + .local Pl$_4$, "iv_length":I 15 + move Pl$_5$, Pl$_4$ 16 + const/16 Pl$_6$, 0x10 17 + add-int/lit8 Pl$_5$, Pl$_5$, -0x10 18 + invoke-virtual { Pl$_3$, Pl$_5$}, Ljava/lang/String;->substring(I)Ljava/lang/String; 19 + move-result-object Pl$_3$ 20 + move-object Pl$_1$, Pl$_3$
	iv	**[Decryption:]** 1 § new-instance Pl$_2$, Ljavax/crypto/spec/IvParameterSpec; 2 − const Pl$_1$, * 3 § invoke-virtual {Pl$_1$}, Ljava/lang/String;->getBytes()[B 4 § move-result-object Pl$_1$ 5 § invoke-direct {Pl$_2$, Pl$_1$}, Ljava/crypto/spec/IvParameterSpec;-><init>([B]V 6 + sget-object Pl$_3$ *Target*;->ivParams:Ljavax/crypto/spec/IvParameterSpec; 7 + invoke-virtual { Pl$_3$}, Ljava/lang/Object;->toString()Ljava/lang/String; 8 + move-result-object Pl$_3$ 9 + invoke-virtual { Pl$_3$}, Ljava/lang/String;->length()I 10 + move-result Pl$_3$ 11 + move Pl$_4$, Pl$_3$ 12 + .local Pl$_4$, "iv_length":I 13 + move Pl$_5$, Pl$_4$ 14 + const/16 Pl$_6$, 0x10 15 + add-int/lit8 Pl$_5$, Pl$_5$, -0x10 16 +invoke-virtual { Pl$_3$, Pl$_5$}, Ljava/lang/String;->substring(I)Ljava/lang/String; 17 + move-result-object Pl$_3$ 18 + move-object Pl$_1$, Pl$_3$

Figure 3.5: [68]

3.4 sPECTRA

sPECTRA is an open source [34] automated framework for detecting cryptographic vulnerabilities in Android applications and it uses hybrid analysis. It is proposed by Gajrani [68] et al. sPECTRA analyzes apk files directly without any alteration to source code. It uses Intelligent UI in the dynamic analysis step for this reason sPECTRA can also analyze obfuscated applications.



```
1. new-instance v7,
   Ljavax/crypto/spec/IvParameterSpec;
2. const-string v10,
   "1234567898765432"
3. invoke-virtual {v10},
   Ljava/lang/String;->getBytes()[B
4. move-result-object v10
5. invoke-direct {v7, v10}, Ljava/crypto
   /spec/IvParameterSpec;-><init>([B)V
```

```
1. new-instance Pl2,
   Ljavax/crypto/spec/IvParameterSpec;
2. const Pl1, *
3. invoke-virtual {Pl1},
   Ljava/lang/String;->getBytes()[B
4. move-result-object Pl1
5. invoke-direct {Pl2, Pl1}, Ljava/crypto
   /spec/IvParameterSpec;-><init>([B)V
```

Actual variable/register	Placeholders
v10	Pl1
v7	Pl2
"1234567898765432"	*

```
1. new-instance v7,
   Ljavax/crypto/spec/IvParameterSpec;
2. invoke-static {}, SecureRandom;
   ->gen_ivParams()Ljavax/crypto/spec/IvParameterSpec;
3. move-result-object v1
4. sput-object v1, Target;
   ->ivParams:Ljavax/crypto/spec/IvParameterSpec;
5. invoke-virtual {v1},
   Ljava/lang/Object;->toString()Ljava/lang/String;
6. move-result-object v1
          ...
7. add-int/lit8 v2, v2, -0x10
8. invoke-virtual { v1, v2},
   Ljava/lang/String;->substring(I)Ljava/lang/String;
9. move-result-object v1
11. move-object v10, v1
12. invoke-virtual v10,
    Ljava/lang/String;->getBytes()[B
13. move-result-object v10
14. invoke-direct {v7, v10},
    Ljava/crypto/spec/IvParameterSpec;-><init>([B)V
```

Figure 3.6: [68]

There are 7000 applications collected from 7 different application stores which are analyzed by this tool and the result shows that 90% of them have cryptographic vulnerabilities.

3.4.1 Rules

In this work, the rules do not mentioned clearly as previous works. Therefore the rules behind emerged by inference. Details of the rules are explained in Chapter 2, according to the rule number defined in the parenthesis.

1. Don't use DES algorithm (56 bits key) (R-32)
2. Don't use ECB mode for encryption (R-01)
3. Don't use AES with key size $\leq$ 128 bits (R-17)
4. Don't use AES CBC/CTR Mode with Static IV (R-03)
5. Don't use Signing Algorithm SHA1withRSA (R-09)
6. Don't use the textbook (raw) algorithm for RSA (NOPADDING) (R-06)

7. Don’t use broken hash functions (MD4, MD5, SHA-1) (R-04)
8. Don’t use fewer than 1,000 iterations for PBE (R-22)
9. Don’t use static salt (related R-03 in Chapter 2)(R-31)
10. Don’t use static key material (R-10, R-29, R-30 and R-31)
11. Don’t use constant seed (R-23)
12. Don’t write data to “/sdcard/*” (R-33)
13. Don’t verify host names in SSL/TLS in trivial ways (R-26)
14. Don’t verify certificates in SSL/TLS in trivial ways (R-27)

In here it is necessary to mention SSL/TLS protocol vulnerabilities with more details since the details in Chapter 2 is not enough in this stage.

SSL/TLS Connection Validation (HostnameVerifier, TrustManager) [68]: Sensitive data is transmitted SSL/TLS protocols to the server. In SSL the following conditions are validated. `CommonName` which is in the certificate shown by the server has to be matched with the `Hostname` of the server. Between the certificate presented by the server and the root CA certificate which is installed on mobile has to have a trust chain.

The `HostnameVerifier` in the `JCA` class validates the two conditions above. However custom implementation of validation methods can cause MitMo(Man in the Mobile) attacks. In the `HostnameVerifier` vulnerability, the return value of `HostnameVerifiers verify()` method is always true. As a result of this even though the `CommonName` does not match the `Hostname`, the return value still returns true as shown in Figure 3.7.

```
HostnameVerifier allHostValid = new HostnameVerifier(){
 public boolean verify(String hname, SSLSession s){
  return true; }};
URL url = new URL("https://www.server.com/");
HttpsURLConnection con = (HttpsURLConnection)url.
      openConnection();
con.setDefaultHostnameVerifier(allHostValid);}
```

Figure 3.7: Vulnerable HostnameVerifier [68]

```
TrustManager tm = new X509TrustManager() {
 @Override
 public void checkServerTrusted(X509Certificate[] chain,
     String authType) {} };
c.init(null, new TrustManager[] { tm }, null);
return c.getSocketFactory();}
```

Figure 3.8: Vulnerable TrustManager [68]

On the other hand in the `TrustManager` vulnerability, all certificates are accepted by the `SSLSocketFactory` even if it is irrespective as shown in Figure 3.8.

3.4.2 Methodology

sPECTRA has 2 phases. Phase 1 makes ready all cryptographic APIs used by the application and the application which uses cryptographic primitives is called potentially sensitive application.

Phase 2 analyzes only sensitive applications with 4 main functions.

- SSL/TLS Vulnerability Identification
- Application Hooking and Repackaging
- Intelligent UI Exploration
- Log Parsing

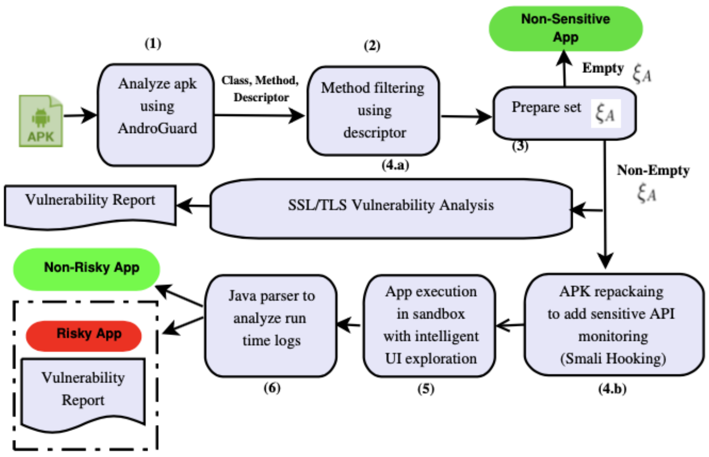

APK
(1)
Analyze apk using AndroGuard
Class, Method, Descriptor
(2)
Method filtering using descriptor
Prepare set ξ_A
(3)
Empty ξ_A
Non-Sensitive App
Non-Empty ξ_A
(4.a)
SSL/TLS Vulnerability Analysis
Vulnerability Report
Non-Risky App
Risky App
Vulnerability Report
Java parser to analyze run time logs
(6)
App execution in sandbox with intelligent UI exploration
(5)
APK repackaing to add sensitive API monitoring (Smali Hooking)
(4.b)

Figure 3.9: Process of sPECTRA

3.4.2.1 Phase 1

The type of parameters which is taken by method and the value which is returned is represented by method descriptor. By using Androguard framework's [2] `get_descriptors` utility, method descriptors are extracted and it is used to filter sensitive applications. The packages `javax.crypto`, `javax.crypto.spec`, `java.security` and `javax.net.ssl` are the most important packages for cryptography. So, the words `crypto`, `security`, `ssl` can be used for filtering the application which contains cryptographic APIs in step 2 of Figure 3.9.

In Figure 3.9, ξ_A is represents the set of all cryptographic sensitive APIs used by the application A. ξ_A in step 3 is built by using the `get_Source()` method of Androguard in the source code for finding actual APIs used.

It is observed that a few vulnerabilities can be checked by using only Phase 1. The remaining types of vulnerabilities and also for obfuscated applications run-time values are needed. Therefore, ξ_A is going to Phase 2.

3.4.2.2 Phase 2

For Phase 2 there are 4 main topics as SSL/TLS Certificate Validation, Application Hooking and Repackaging, Intelligent UI Exploration, Log Parsing.

1. **SSL/TLS Certificate Validation:** In this module, the type of vulnerability can be detected by using static analysis. Analysis bases Soot library [63] and this library have tagging features as "Vulnerable" or "Non-Vulnerable". Points-To analysis, CFG and Data-Flow analysis features are used. CFG is mentioned in Chapter 3.1. Point-To analysis is a static analysis technique, that is, let $s = a$; $s = b$, then s can point to a or b in execution time. Thus, Points-To analysis s is $Points - To(s) = a, b$.

   sPECTRA firstly creates IR (intermediate representations) in the form of CFG. Secondly, it sets Point-To analysis and lastly check following on IRs;

   - Exit nodes in CFG which have classes containing the `HostnameVerifier` interface are analyzed in this module. If exit nodes always return a true value, it means that there is a vulnerability.
   - Points-To set of `SSLSocketFactory` is calculated, to determine whether the `HostnameVerifier` is vulnerable or not. If the set contains `AllowAllHostnameVerifier`, then it is tagged as vulnerable.
   - The lack of custom implementations of `X509TrustManager` are also a vulnerability for sPECTRA.

2. **Application Hooking and Repackaging:** Monitoring code contains returned values of an API invocation and logs run-time parameters. By using APIMonitor [4], monitoring code of an API, which is in ξ_A, can be added to an application. After monitoring code is added to an application, apk is repackaged and analyzed in an emulator.

3. **Intelligent UI Exploration:** User interaction simulating is a crucial step while detecting vulnerability in dynamic analysis based systems. Monkey [23] and MonkeyRunner [24] tools can be used in exploration. But these tools are affected by the objects' coordinates. Thus, small changes in location of view can generate errors. On the other hand, sPECTRA has a generalized system which is not affected by the objects' position. That is sPECTRA uses their own UI Exploration system.

4. **Log Parsing:** This module collects logs during the execution in an emulator by using the logcat utility of adb and finds the vulnerabilities comprehensively from these logs.

3.5 BinSight

BinSight [79] is a static cryptographic misuse detection tool. By using BinSight 132K Android application evaluating which was collected in 2012, 2015 and 2016 [79]. The evaluation shows that application and libraries usage rates of ECB mode and the static seeds for `SecureRandom` class have decreased but, libraries have increased the use of static encryption keys and static IVs in CBC mode. Moreover, insecure RC4 and DES ciphers have become the second and third mostly used ciphers in 2016.

CryptoLint has a limitation about whether a misuse happens because of an application code or a third-party library. Although BinSight takes as a basis to CryptoLint, it can overcome this limitation. Moreover, although CryptoLint cannot analyze 23% APK files of the analyzed set as mentioned in Chapter 3.1, BinSight cannot analyze only 0.005% APK files .

3.5.1 Rules

BinSight uses the CryptoLint six rules. Details of the rules are explained in Chapter 2, according to the rule number defined in the parenthesis.

1. Do not use ECB mode for encryption (R-01)
2. Do not use a non-random IV for CBC encryption (R-03)
3. Do not use constant encryption key (R-10)
4. Do not use constant salts for PBE (R-20)
5. Do not use fewer than 1,000 iterations for PBE (R-22)
6. Do not use static seeds to seed `SecureRandom` (R-23)

3.5.2 Methodology

BinSight based on CryptoLint methodology. In addition to the methodology of CryptoLint, BinSight has a graphical UI and uses source attribution for manual analysis. Furthermore, there are 2 new steps to apk analysis to deal with mentioned limitations above of CryptoLint. The procedure of the BinSight is shown in the Figure 3.10.

1. **Preprocessing:** This phase consists of 2 parts.
   (a) **Disassembly:** apk files are disassembled by using apktool [5] into smali files. After disassembling, the tool finds the entry point of cryptographic APIs in smali files.

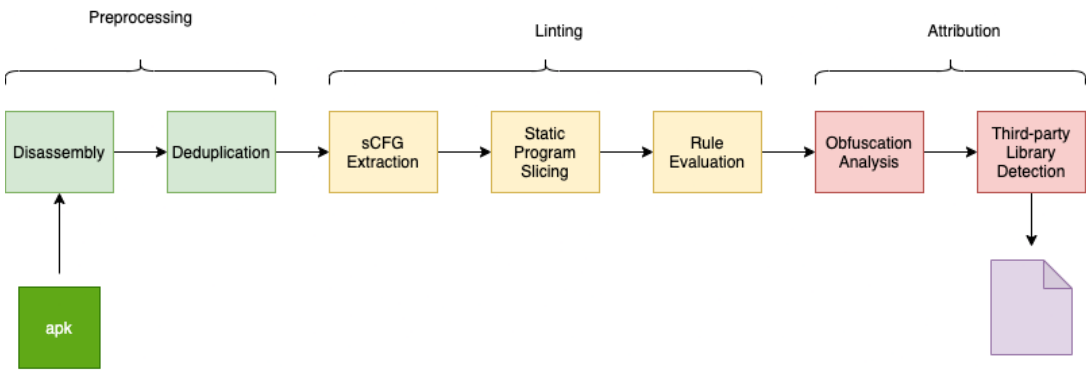


Figure 3.10: Process of BinSight

(b) **Deduplication:** Since a huge amount of applications are downloaded from Google Play, there are possible duplicates in the dataset. Every APK file has an ID value which is in the Android manifest file. By using these ID values and download time of an application, the latest version of an application is kept in the dataset.

2. **Linting:** BinSight computes static program slicing which terminates in calls to cryptographic APIs and it gets the needed information at the end of these slices. As a result of this, it determines whether rules mentioned in Chapter 3.5.1 are violated or not.

   (a) **sCFG Extraction:** It is the same procedure with CryptoLint as mentioned in Chapter 3.1.2.

   (b) **Static Program Slicing:** BinSight uses static program slicing on the sCFG to determine whether the analyzed application uses cryptographic APIs or not. That is, BinSight explores the sCFG for nodes belonging to Java cryptographic APIs endpoint. When this type of nodes are found, BinSight uses incoming edges to address all call sites in the application.

   (c) **Rule Evaluation:** The value of parameters of cryptographic API call is the base of rule evaluation. BinSight uses a backward slice for detecting how a parameter changes until it finds the first assignment of a parameter. And the first parameter can be static.

3. **Attribution:** The attribution phase needs to handle obfuscated package names to map call sites to libraries.

   (a) **Obfuscation Analysis:** To determine the level of obfuscation, a rule based classifier is used. Rule based classification has 7 rules [66];

   - If all parts of the identifier are of length one, then this case is full obfuscation.
   - If all but the first part of the identifier are of length one and the first part is in the set $\{com, ch, org, io, jp, net\}$, then this case is partial obfuscation.
   - If none of the package name parts in the identifier are of length one, then this case is either none or class-level obfuscation.
   - if at least one part but not all of the identifiers are of length one, then the case is partial obfuscation.

- If the class name is longer than 3 chars then it is no obfuscation.
- If class name length is 1 character, then this case is class obfuscation.
- If class name of length 2 or 3 characters and the first character is in lowercase, then this is class obfuscation.
  BinSight can automatically detect the level of obfuscation. Unless the package is fully obfuscated, class name is used to identify the library it belongs to.

(b) **Third-party library detection:** All call sites which terminate cryptographic APIs have distinguishable package names except obfuscated ones. For this step it is needed to decide a package name corresponding to an application, library or possible library from the third party.

3.6 CHIRON

CHIRON is a static analysis tool introduced by Rahaman et al. [85]. The aim is to produce deployment-quality program which can detect misuses in cryptographic API in Java.

CHIRON differs from previous tools since it has low false alerts rates. CHIRON can remove irrelevant elements in program slicing which cause false alerts, decreases the number of misuse detections and adds new security rules. These new rules are SSL/TLS vulnerabilities, usage of weak crypto primitives, misuses about pseudorandom number generators and keystores. Besides backward program slicing, CHIRON has intra and inter-procedural forward program slicing.

They evaluate CHIRON with 46 Apache projects and 240 apk files.

3.6.1 Rules

Details of the rules are explained in Chapter 2, according to the rule number defined in the parenthesis.

1. Don't use predictable/constant cryptographic keys (R-10, R-11)
2. Don't use predictable/constant passwords for PBE (R-20)
3. Don't use predictable/constant passwords for KeyStore (R-16)
4. Don't use custom Hostname verifiers to accept all hosts (R-26)
5. Don't use custom `TrustManager` to trust all certificates (R-27)
6. Don't use custom `SSLSocketFactory` w/o manual `Hostname` verification (R-26, R-28)

7. Don't occasionally use HTTP (R-25)
8. Don't use predictable PRNG seeds (R-23)
9. Don't use cryptographically insecure PRNGs (e.g., `java.util.Random`) (R-24)
10. Don't use static Salts in PBE (R-20)
11. Don't use ECB mode in symmetric ciphers (R-01)
12. Don't use static IVs in CBC mode symmetric ciphers (R-03)
13. Don't use fewer than 1,000 iterations for PBE (R-22)
14. Don't use 64-bit block ciphers (DES, IDEA, Blowfish, RC4, RC2, ...) (R-03)
15. Don't use insecure asymmetric ciphers (R-05, R-06, R-07, R-08 and R-09)
16. Don't use insecure cryptographic hash (e.g., SHA1, MD5, MD4, MD2) (R-04)

3.6.2 Methodology

CHIRON uses both backward and forward slicing to detect cryptographic vulnerabilities. First of all, definitions of def-use analysis, backward and forward program slicing will be given, slicing criteria is defined in Chapter 3.1.2. Def-use analysis declares the usage of statements and expresses their dependency relations. Backward program slicing is to calculate a set of program statements which affect the slicing criterion in relation to data flow, for a slicing criterion and forward program slicing is to calculate a set of program statements which are affected by the slicing criterion in relation to data flow, for a slicing criterion. Figure 3.11 shows the workflow of the CHIRON.

1. **Backward Program Slicing:**
   (a) **Intra-procedural backward slicing** has 2 main features. One of them is that it is a building block of inter-procedural backward slicing. The other is that it uses def-use method to determine whether a statement is in a slice or not.
   (b) **Inter-procedural backward slicing** depends on intra-procedural backward slicing. Caller-calle graph of every method in the program is designed. All method invocation in the specified method is determined in the slicing criterion. `calcInterProceduralSlices` computes all the inter-procedural backward slices for all method invocation.
2. **Forward Program Slicing:** Previous tools based only backward program slicing. CHIRON uses forward slicing in addition to backward slicing as a new technique.
   (a) **Intra-procedural forward slicing** is used to check rule 6 and rule 15 and it is nearly the same as intra-procedural backward slicing. Assignments are chosen as slicing criteria. In forward slicing direction is order of the execution.

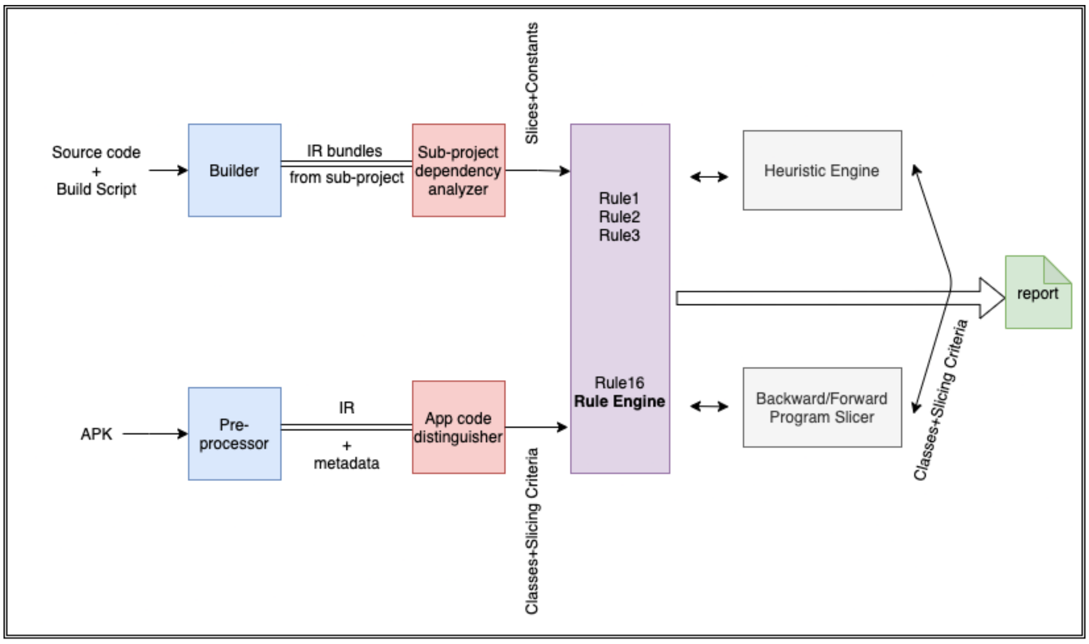


Figure 3.11: Process of CHIRON

(b) **Inter-procedural forward slicing** uses an assignment or a constant as a slicing criterion and it detects the instructions which are affected by the slicing criterion in relation to def-use relations. The used slices in the inter-procedural forward slicing is come from inter-procedural backward program slicing.

3.6.2.1 Heuristics

A tool is referred to as a deployment-quality tool when it has a low rate of false positive alerts. CHIRON is a deployment-quality tool and it has a set of heuristic algorithms to decrease the rate of false positive alerts.

Rule 1, 2, 3, 8, 10, 12, 13 and 15 need to detect constants in a program slice. CHIRON's heuristic algorithm intends to detect hardcoded data or data derived from hardcoded data. But some of the constant data do not impact on security, this kind of data is called pseudo-influences and it is the main cause of false alerts. There are 5 strategies to remove pseudo-influences from the slices.

1. **Heuristic I Removal of state indicators:** It eliminates the constants which describes the state of a variable.

2. **Heuristic II Removal of resource identifiers:** It eliminates the constants which are identifiers of a value source.

3. **Heuristic III Removal of bookkeeping indices:** It eliminates the constants which are indices or sizes of a data structure.

4. **Heuristic IV Removal of contextually incompatible constants:** It eliminates the constants when their types are incompatible with the analysis context.

5. **Heuristic V Removal of constants in infeasible paths:** It eliminates the constants which are initializations and do not have a valid path of influence to the criterion.

3.7 CryptoGuard

CryptoGuard [84] is the same tool as CHIRON. In the paper there are some changes, but they do not affect the rules based on detecting misuses and the methodology behind the tool. The changes are in the evaluation part. CryptoGuard compares with more tools and it is open source while CHIRON is not. Moreover the writer team changes from 2018 to 2019.

3.7.1 Rules

The rules which are used are the same with CHIRON [85].

3.7.2 Methodology

The methodology behind Cryptoguard is also the same as the methodology of CHIRON [85].

3.8 CRYLOGGER

CRYLOGGER is the first dynamic analysis tool which is open source [10, 83]. It is used for Android and Java applications. CRYLOGGER logs the parameters of the invoked APIs and determines whether these parameters comply with the cryptographic rules or not.

CRYLOGGER consists of 26 rules and to facilitate adding new rules, the checking and logging parts are decoupled.

3.8.1 Rules

Details of the rules are explained in Chapter 2 according to the rule number defined in the parenthesis.

1. Don't use broken hash functions (SHA1, MD2, MD5, ...) (R-04)

2. Don't use broken encryption alg. (RC2, DES, IDEA, ...) (R-32)

3. Don't use the operation mode ECB with > 1 data block (R-01)

4. Don’t use the operation mode CBC (client/server scenarios) (R-02)
5. Don’t use a static (= constant) key for encryption (R-10)
6. Don’t use a “badly-derived” key for encryption (R-11)
7. Don’t use a static (= constant) initialization vector (IV) (R-29)
8. Don’t use a “badly-derived” initialization vector (IV) (R-30)
9. Don’t reuse the initialization vector (IV) and key pairs (R-12)
10. Don’t use a static (= constant) salt for key derivation (R-20)
11. Don’t use a short salt (< 64 bits) for key derivation (R-21)
12. Don’t use the same salt for different purposes (R-31)
13. Don’t use < 1000 iterations for key derivation (R-22)
14. Don’t use a weak password (score < 3) [78] (R-13)
15. Don’t use a NIST-black-listed password [26] (R-14)
16. Don’t reuse a password multiple times [26] (R-15)
17. Don’t use a static (= constant) seed for PRNG [54] (R-23)
18. Don’t use an unsafe PRNG (java.util.Random) [54] (R-24)
19. Don’t use a short key (< 2048 bits) for RSA [63] (R-05)
20. Don’t use the textbook (raw) algorithm for RSA [30] (R-06)
21. Don’t use the padding PKCS1-v1.5 for RSA [86] (R-07)
22. Don’t use HTTP URL connections (use HTTPS) [79] (R-25)
23. Don’t use a static (= constant) password for store [26] (R-16)
24. Don’t verify host names in SSL in trivial ways [79](R-26)
25. Don’t verify certificates in SSL in trivial ways [79] (R-27)
26. Don’t manually change the hostname verifier (R-28)

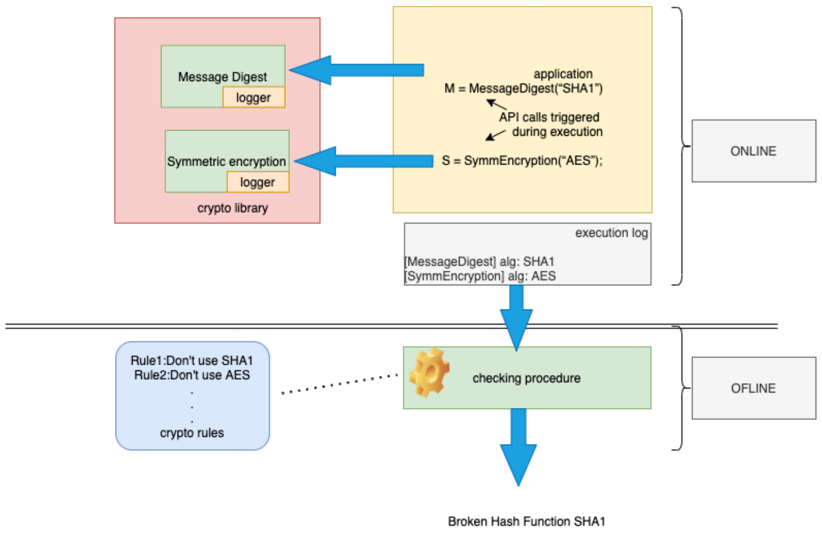


Figure 3.12: Process of CRYLOGGER

3.8.2 Methodology

CRYLOGGER consists of logger and checker parts. In the logger part relevant parameters are logged which is necessary to check cryptographic rules. For instance, the names of the algorithms such as SHA1 and AES that are chosen by the application are saved via logs. In the checker part the application is executed firstly and the checker analysis logs offline to reduce the impact on the application performance. Then, it shows the list of cryptographic rules violated. The process of the CRYLOGGER is illustrated in Figure 3.12.

There are 4 main procedures to check the rules defined in the previous section and shown in Figure 3.13. They are unacceptable values, constant values, badly-derived values and reused values.

1. **Unacceptable values:**

   All the values extracted from the logs are sent to a rule specific function to determine whether the values comply with rules or not according to the yes/no answers. This procedure can be used in most of the rules. For instance, for Rule 1 the value `alg` of `MessageDigest` cannot be SHA1, MD2 and MD5 and similarly the value of `alg` of `SymmEncryption` cannot be broken algorithms like DES.

$$\text{Rule1: } \texttt{MessageDigest.alg} \notin \{\text{"}SHA1\text{"}, ...\}$$
$$\text{Rule2: } \texttt{SymmEncryption.alg} \notin \{\text{"}DES\text{"}, ...\}$$

   For Rule 3 and Rule 4, just 1 data block usage of CBC is acceptable otherwise the modes of ECB/CBC cannot be used.

$$\text{Rule3: } \texttt{SymmEncryption.mode} \neq \text{"}ECB\text{" or}$$
$$\texttt{SymmEncryption.}\#\text{blocks} = 1$$
$$\text{Rule4:}\texttt{SymmEncryption.mode} \neq \text{"}CBC\text{" or}$$

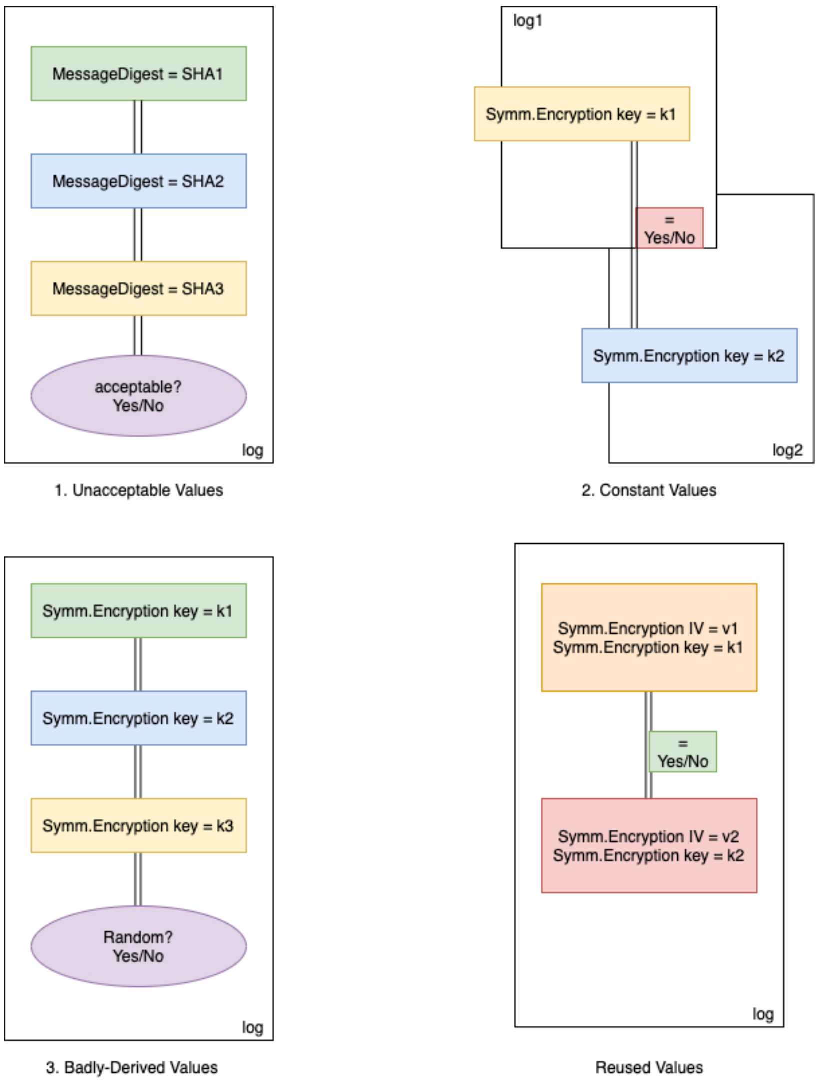


Figure 3.13: 4 main procedure of CRYLOGGER

The length of salt should be more than or equal to 64 bits and the number of the iteration should be more than or equal to 1000 for the key derivation.

Rule 14: $\texttt{KeyDerivation.pass} \notin$ BadPass
Rule 15: score $\texttt{(KeyDerivation.pass)} \geq 3$

To say that a random number generator is secure, generating numbers must be truly-random. For instance, `java.secure.SecureRandom` has to be used in place of `java.util.Random`.

Rule 18: $\texttt{RandomGenerator.alg} = \mathit{Secure}$

The encryption key of RSA has to be greater or equal to 2048 bits and it is not acceptable `NOPADDING` and PKCS-v1.5 for padding.

Rule 19: $\texttt{AsymmEncryption.alg} \neq RSA$ or

`AsymmEncryption.key` $\geq$ 2048 bits
Rule 20: `AsymmEncryption.alg` $\neq RSA$ or
`AsymmEncryption.pad` $\neq$ `NOPADDING`
Rule 21: `AsymmEncryption.alg` $\neq RSA$ or
`AsymmEncryption.pad` $\neq$ `PKCS1-v1.5`

The connection protocol cannot be HTTP.

Rule 22: `SSL/TLS/Cert.urlprot` $\neq HTTP$

It is necessary to check whether applications trust all certificates and hosts for Rule 24 and Rule 25. Whether the default hostname verifier is replaced to avoid hostname verification or not, for rule Rule 26.

Rule 24: `SSL/TLS/Cert.allhost` $= False$
Rule 25: `SSL/TLS/Cert.allcert` $= False$
Rule 26: `SSL/TLS/Cert.sethost` not assigned

2. **Constant Values**

   This procedure is used for Rule 5, Rule 7, Rule 10, Rule 17 and Rule 23 rules and checks parameters are constant or not. For example, static encryption keys must not be used and keys have to be generated by a random generator. To determine whether constant values are used or not, two execution logs are necessary. The values found in one log cannot be in second. $\{\}_1$ is shown the values which are in the first log and $\{\}_2$ is shown the values which are in the second log and the following property has to be hold;

   Rule 5: $\{SymmEncryption.key\}_1 \bigcap \{SymmEncryption.key\}_2 = \varnothing$

3. **Badly-derived Values**

   This procedure is about whether the value is truly-random or not. In Rule 6, encryption keys have to be truly-random. The following three checks are necessary to determine level of randomness.

   (a) If the value `alg` of `RandomGenerator` is `Secure`, then it is legit value.
   (b) If the value `alg` of `RandomGenerator` is not `Secure`, then it is a bad value.
   (c) If the previous conditions do not hold, NIST tests for randomness [86] are used, and the value is called bad when at least one test fails.

   This procedure is used for Rule 8 also. Static approaches do not contain this procedure.

4. **Reused Values**

   This procedure determines whether a value is reused in executions of an application. In Rule 9, it is necessary that the key pair (key, IV) is never used twice. The procedure checks the logs to detect duplicates.

Rule 9: `containsDuplicates({(SymmEncryption.key, SymmEncryption.iv)})` $= False$

This procedure is used for Rule 12 also. Static approaches do not contain this procedure.

CHAPTER 4

DETECTION TOOLS FOR IOS APPLICATIONS

iOS applications have the ipa file format which contains binary files that allow them to be installed on iOS or ARM-based MacOS devices. On the contrary to the open source Android platform, the iOS platform is relatively close and it makes their analysis hard. For example, Android applications have a bytecode format which is reversible, but iOS applications can be compiled to a machine code which is custom for a specific CPU architecture. For this reason there is limited work that is capable of detecting cryptographic misuse for the iOS platform.

In the iOS platform, `CommonCrypto` and `Security` libraries invoke security related APIs. `CommonCrypto` library is used for hash functions, symmetric ciphers and key derivation functions. `Security` library is used for asymmetric ciphers, access of keychain and handling of certificates.

4.1 iCryptoTracer

iCryptoTracer is a tool which combines static and dynamic analysis to detect cryptographic misuses for iOS applications [76]. iCrypto Tracer follows the cryptographic APIs of the iOS application, reveals the trace logs and determines whether the application complies with the cryptographic rules or not.

By using iCryptoTracer, 98 iOS applications are analyzed and 68 of them have several types of cryptographic misuses.

4.1.1 Rules

The rules are defined as **Items** in this article of this tool and there are 3 items behind the mechanism of the tool.

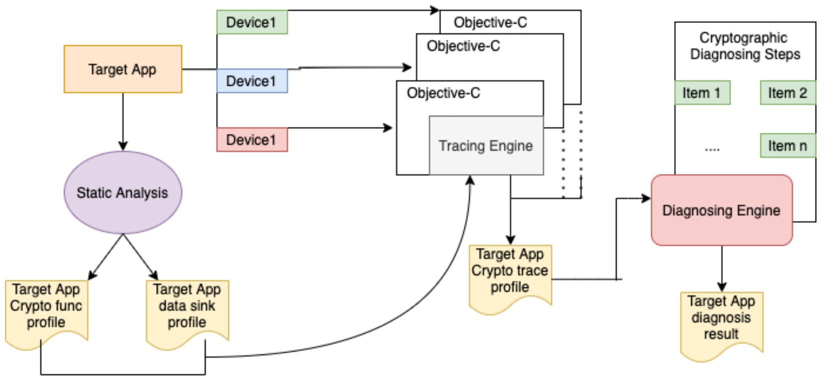
Item1 Using constant encryption keys (R-10)

Item2 Using a non-random IV for CBC encryption(R-03)

Item3 Using stateless encryption: In Common Crypto library there are 2 encryption interfaces such as `Stateless` and `Stateful` ones. Deterministic encryption means that if the same plaintext encrypts with the same key two times, the ciphertexts will be equal and it means that an attacker can detect the repeats. For this reason deterministic encryption is insecure. Since the stateless encryption is deterministic, stateless encryption is also insecure.

4.1.2 Methodology

iCrytoTracer analyzes the iOS applications by combining static and dynamic techniques. The methodology of iCryptoTracer is shown in Figure 4.1.

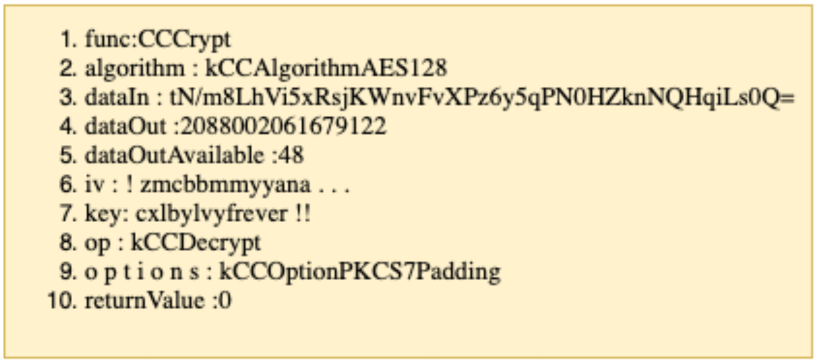

Target App
Device1
Device1
Device1
Objective-C
Objective-C
Objective-C
Tracing Engine
Cryptographic Diagnosing Steps
Item 1
Item 2
....
Item n
Diagnosing Engine
Static Analysis
Target App Crypto func profile
Target App data sink profile
Target App Crypto trace profile
Target App diagnosis result
Figure 4.1: Process of iCryptoTracer

iCryptoTracer first scans the application to find all data sink by using static analysis. Second, it scans the application to detect cryptographic APIs at this time. By using these two steps, the cryptographic function profile and data sink profile are generated. Related functions calls are redirected to the tracing engine to log all valuable information such as sequence of API calls, return values etc. to the crypto-trace file. In the last step, information which is gathered is sent to the diagnosis engine to determine whether a cryptographic misuse exists or not.

In the static analysis phase, iOS applications are downloaded from the Apple Store. Afterwards, APIs which relate to the data operations, transmissions and encryption are filtered. Then the locations of APIs are resolved into files cryptographic function profile and data sink profile. But static analysis is not enough to get all essential information. Thus, in the runtime, more exact information is collected.

Relevant API information is stored into a log file. An example of a log file is in Figure 4.2. This step is called Log Tracing.

After the log tracing step, the logs are sent to the diagnosis engine. Logs are collected from different devices with different versions of the operating system for the same app. Then the diagnosis engine detect the misuses depending the items defined in Chapter 4.1.1. After the diagnosis procedure, the diagnosis result file arises. The diagnosis result file contains the security degree of the application (critical, weak, healthy), location of suspicious encryption functions and their relevant data sink. Security degree of an application shows in Table 4.1.



```
func:CCCrypt
algorithm : kCCAlgorithmAES128
dataIn : tN/m8LhVi5xRsjKWnvFvXPz6y5qPN0HZknNQHqiLs0Q=
dataOut :2088002061679122
dataOutAvailable :48
iv : ! zmcbbmmyyana . . .
key: cxlbylvyfrever !!
op : kCCDecrypt
o p t i o n s : kCCOptionPKCS7Padding
returnValue :0
```

Figure 4.2: An example of a log file

Table 4.1: Security degree of an application

	Pass all the item control	Happen at data sink
Healty	Yes	-
Weak	No	No
Critical	No	Yes

4.2 Automated Binary Analysis on iOS

In this work[65], a framework is developed which can decompile 64-bit ARMv8 binaries to LLVM IR code for iOS applications. This framework facilitates analyzing iOS applications and constructing control and data flow can be modelled to detect cryptographic misuses. Since the framework does not have a name, it is mentioned as the framework.

According to the analysis using this framework, 343 iOS applications out of 417 violate at least one cryptographic rule.

4.2.1 Rules

These tools use the six rules of CryptoLint[61] by integrating the rules in *CommonCrypto* library which is default cryptographic library in iOS. The details about the six rules are explained here since there are some differences because the system is iOS. For further information about the rules see Chapter 2, according to the rule number defined in the parenthesis.

1. Do not use ECB mode for encryption (R-01)
2. Do not use a non-random IV for CBC encryption (R-03)
3. Do not use constant encryption keys (R-10)
4. Do not use constant salts for PBE (R-20)
5. Do not use fewer than 1,000 iterations for PBE (R-22)
6. Do not use static seeds to seed SecureRandom (R-23)

4.2.2 Methodology

In Figure 4.3, the workflow of the detection process of the framework is shown. Raw iOS binary is took as an input. Then the following steps are followed;

1. **Disassemble:** 64-bit ARMv8 binary is extracted in a Mach-O file and the disassembler of the LLVM framework is configured to create assembly code.
2. **Decompile:** In this phase, 64-bit ARMv8 binaries which are extracted from the disassemble part are translated LLVM IR code to obtain basic representation.
3. **Pointer Analysis:** This part is necessary to obtain the next slicing step with the information about pointers. Moreover, in this part call graphs are generated which is essential for data flow analysis.
4. **Static Slicing:** As mentioned in the earlier Chapter 3.1, static slicing aims to find all code slices which affect the slicing criterion. By using static slicing the code slices which related to cryptographic rules in Chapter 4.2.1 are detected.

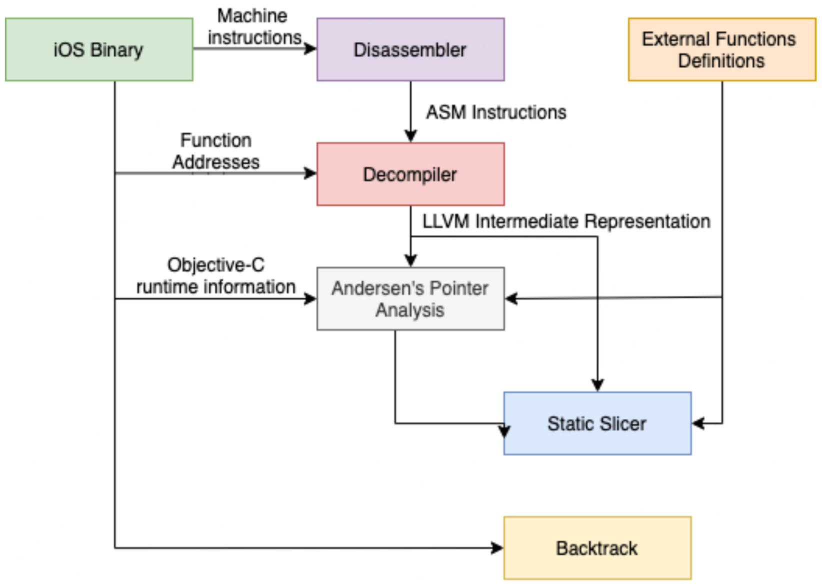

iOS Binary
Machine instructions
Disassembler
External Functions Definitions
ASM Instructions
Function Addresses
Decompiler
LLVM Intermediate Representation
Objective-C runtime information
Andersen's Pointer Analysis
Static Slicer
Backtrack

Figure 4.3: Process of Framework

5. **Parameter Backtracking:** After static slicing, to decide whether a parameter complies with the cryptographic rules as defined before or not, parameter backtracking is used. According to the slicing criterion, each execution path a parameter can take is backtracked and if there exists a rule violation, the information flow which is affected can be detected.

CHAPTER 5

DETECTION TOOLS FOR IOT

Firmwares which gives the low level control of the specific hardware of a device is a specific class of computer software. It runs on embedded devices with specific purposes and these embedded devices are different from each other according to architectures.

In this platform lots of the developers use their own cryptographic APIs for optimization. For this reason, to detect these self-defined cryptographic APIs, `libcrypto`, `libcrypt`, `cryptlib`, `Nettle`, `libgcrypt`, `wolfcrypt` and `LibTomCrypt` can be used as starting points.

5.1 CRYPTOREX

CRYPTOREX is a framework developed to detect cryptographic misuse in IoT devices [94]. Up to this, existing ones can detect only the misuse on the Android, Java and IOS platforms and there is no tool for IoT devices. Since the IoT applications involve different architecture, they do not have reversible bytecode format and it is the reason for the lack of the detection mechanism for IoT devices.

By using 165 defined cryptographic APIs, an analysis is conducted on 521 firmware and 419 cryptographic misuse is detected. It means that 24% of the firmware violates the cryptographic rules.

5.1.1 Rules

CRYPTOREX also used the same rules as CryptoLint. For further information about the rules see Chapter 2, according to the rule number defined in the paranthesis.

1. Do not use ECB mode for encryption (R-01)
2. Do not use a non-random IV for CBC encryption (R-03)
3. Do not use constant encryption keys (R-10)

4. Do not use constant salts for PBE (R-20)
5. Do not use fewer than 1,000 iterations for PBE (R-22)
6. Do not use static seeds to seed `SecureRandom` (R-23)

5.1.2 Methodology

The methodology contains five steps. CRYPTOREX takes raw firmware images as an input and gives the report which gives detailed information about the misuse of the cryptographic functions used in the IoT applications. The procedure is shown Figure 5.1.

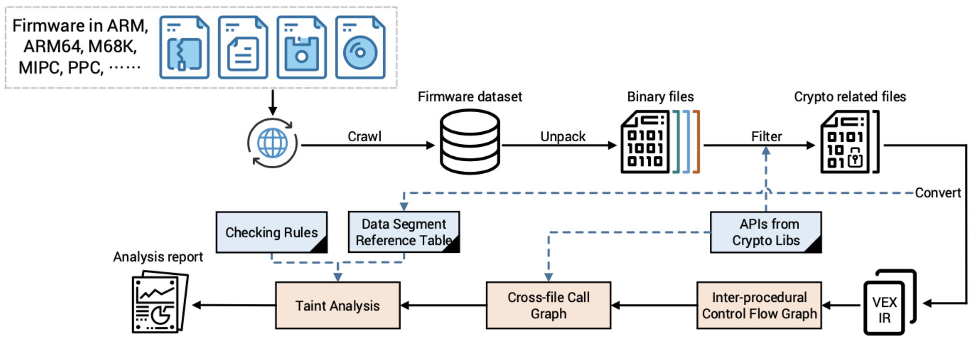


Figure 5.1: Process of CRYPTOREX

1. **Firmware Acquisition and Pre-processing:** Crawler is a program which browse the Internet systematically to collect data. To download firmware images from the Internet automatically, a crawler is developed. But, the firmwares are packed with different compression algorithms and it is not possible to analyze directly. For this reason to unpack the firmwares, Binwalk [8] is used.

2. **Lifting to VEX IR:** Firstly, file filter is used to reduce time consumption in the conversion of IR transformation. The header information of each binary file is checked to determine cryptographic APIs used or not. If there is no cryptographic API used in a binary file, then it can be ignored.

   The VEX IR representation format is used [81]. After the file filter process, CRYPTOREX uses the Angr [3]to disassemble the binary file, then VEX IR is lifted. However, the transformation of binary to IR is not sufficient. Since;

   - If the address of a call function is put into the memory, Angr cannot find it since Angr can locate only the explicit invocation address.
   - The type information of a variable can be lost and it affects the capture data flow.
   - The argument of a function can pass in the register or stack.

   To fix the above first two cases; a script for IDA Pro [15] is developed. And for the last one, the arguments which pass rules of different architecture are extracted.

3. **Inter-procedural Control Flow Graph Construction(ICFG):** ICFG is essential to generate intra and inter procedural data-flow analysis. ICFG starts with an entry point for an executable and it goes through the functions. Function call relation is used to support inter-procedural analysis. Isolated functions are also discovered and therefore function call relations can be discovered as many as possible. Isolated functions are important because library files only use isolated files as a function. As mentioned before Angr can detect explicit function addresses. But, recover the addresses of indirectly invoked functions, previous instructions are simulated and the value of the register or memory is calculated. If this value defines an address, it is added to the function call relations.

4. **Cross-file Call Graph Construction:** Developers of IoT can use the self defined APIs for optimization. For this reason, executable files can invoke the original API or self-defined ones. It is necessary to add self-defined APIs to the function call relation graphs. Thus, Cross-file Call Graph Construction is used which include multiple executable files and their function call relation for each cryptographic API. By using this construction, the cryptographic API list can be updated and the cryptographic misuse of self-defined API can be detected.

5. **Taint Analysis:** This step aims to detect whether there exists a cryptographic misuse or not in the IoT applications. To do that firstly, the arguments of cryptographic APIs, i.e sources, are determined. Secondly, the wrappers of cryptographic APIs are determined and based on the way function arguments propagated, API lists are updated. Lastly, in the sink phase, the misuse rules are applied.

   (a) **Taint Sources:** To tag the taint sources, it is necessary to determine that the function's argument corresponds to the function parameter which is related to the cryptographic misuse. To do that the arguments which are the passing rule of the tested firmware are matched and recorded in the second step (lifting VEX IR).

   (b) **Taint Propagation:** The use-defined chain algorithm of the Angr is used for building data dependence. But, it gives false negative results in the array operation APIs in C. To handle it, a module which simulates the array operations is created. Moreover, the list of cryptographic APIs are updated during taint analysis. The functions (self-defined cryptographic API) whose parameters are passed to the cryptographic API as an argument are added to the CFCG.

   (c) **Taint Sinks:** At constants, taint sinks are defined. There are two possible inferences for a constant. It can be either a pointer or an immediate value according to the types of function argument which is in the function prototypes. In case the constant is a pointer, its referenced data can be reached in the data segment reference table. For the latter case that is constant is an immediate value, there are no further steps. After deciding the type of constant, it is necessary to determine whether data is matched with the specifications defined in misuse rules. If data is matched, then it is called a violation and reported in the report.

CHAPTER 6

SHORTCOMINGS OF TOOLS

There are two types of misuse detection approaches, static and dynamic. As mentioned before, both of them have some cons and pros. In the static approach, the analyzed program does not need to be executed. Since execution needs significant effort to trigger and detect misuses, it limits the capability of covering a wide range of security rules. Thus static approaches can check more security rules than the dynamic approaches. On the other hand, lack of execution in the static approaches can cause more tendency to produce false alerts. False alerts can be categorized into two groups.

- **False positive:** False positive means that a tool gives an alert unexpectedly. That is, even though there is no misuse in the test case, the tool raises an alert.
- **False negative:** False negative means that there exists a misuse, but a tool cannot detect it.

These kinds of alerts reduce the value of the static misuse detection tools like CryptoLint [61], CDRep [77], BinSight [79], CryptoGuard [84], Automated Binary Analysis on iOS [65] and CRYPTOREX [94]. Moreover, static analysis tools cannot detect cryptographic misuses in the obfuscated codes or codes which have some run-time dependency. Thus they can miss some cryptographic misuses which occur after loading dynamically.

Besides the common shortcomings of static analysis, each static analysis tool has other shortcomings. CryptoLint can analyze only the Dalvik bytecode of an application. However, some applications can use the native code also for their cryptographic operations. Due to the nature of cryptographic, some parameters have to be non-predictable and unique to make secure cryptographic systems. But, CryptoLint can only check whether these parameters are generated from static value. It cannot detect if the same values are used in different executions or not. CDRep [77] can detect only cryptographic misuses in the Android API, it cannot detect the misuses in the third-party library or in the library which is implemented by the application developers. Moreover, except CDRep, none of the detection tools cannot fix the misuses. BinSight [79] and CHIRON [85] (more commonly known as CryptoGuard [84]) analyze every path in the application even though it is not executed in reality. For this reason they can raise false positive or false negative alerts like other static tools. Automated Binary Analysis on iOS [65] is one of them and it uses parameter backtracking. However, some call graphs have lots of edges and it has a negative impact on backtracking accuracy. Also, like user interface events, external libraries cannot be resolved completely by using this tool and it results with incomplete call graphs. Consequently, the data flow between functions cannot entirely be modeled and parameter backtracking becomes infeasible. CRYPTOREX [94] is the other. In this framework, to obtain executables, target firmware is unpacked. This unpackage step runs with BinWalk. But, if the firmware is packed with the custom compression algorithm, BinWalk cannot unpack them. Furthermore, CRYPTOREX can check only seven cryptographic libraries. Even though it covers nearly all cases, some developers can use their non-standard cryptographic implementations and like the other static approaches CRYPTOREX cannot detect them.

Like static approaches, dynamic approaches have some negative aspects. First of all it can check less security rules. Also theoretically, there are no false positives in dynamic analysis, but in practice there are. Dynamic analysis executes the programs to detect misuses and does not give any information about which part has to obey the security rules to protect secret data. It just shows the misuses. But, in some situations there is no need for correct usage of cryptographic. For example, if there is no sensitive data, it is not necessary to use cryptographic correctly. Thus to determine if there is sensitive data in a location of the code, manual analysis is required. Also, some cryptographic misuses can be missed, if they are not triggered in the execution. To the best of our knowledge CRYLOGGER [83] is the first and only dynamic analysis tool and it inherits all the negative aspects of the dynamic approach.

Because the static and dynamic approach have some cons, hybrid analysis tools are developed like CMA [87], iCryptoTracer [76] and sPECTRA [68]. CMA is a hybrid analysis but it runs manually. So, CMA cannot work on large sets and like CryptoLint it cannot analyze native code. iCryptoTracer can detect cryptographic misuses on a limited set of APIs. Moreover, iCryptoTracer is not able to diagnose misuses in the third-party libraries. sPECTRA can detect cryptographic API in the application whose package name contains `crypto`, `security` and `ssl`. It limits the detection of cryptographic API usage and consequently detection of the existing vulnerability in the code.

According to research [57, 59], static and dynamic approaches are not sufficient alone to detect cryptographic misuse. Using a combination of them gives more accurate results. It is necessary to complement a dynamic tool with a static tool or a static tool with a dynamic tool. To make up for the deficiency about complementation, hybrid approaches are used. But, as mentioned above, hybrid tools also have some negative sides. For this reason, to detect cryptographic vulnerabilities, more than one tool and approach usage is a more effective method.

CHAPTER 7

RELATED WORKS

7.1 Detection of Crypto Misuses on Android and Java Platform

Although there exists lots of recommendations of the official documents and the related works of misuses in Android applications, developers still can make mistakes in their code which makes their applications vulnerable against several attacks. FixDroid [82] is a static tool that can detect vulnerabilities in Android applications. Moreover, it can teach developers to write secure code and improve application security. By using this tool, developers can make quick fixes by using real-time feedback. On the other hand, according to Nadi et al. [80], many developers find the usage of cryptographic APIs is complex to use since they are not cryptographers. CRYSL [71] is a definition language to bridge the gap between developers and cryptographers. Cryptographers can specify the correct usage of cryptographic libraries by using CRYSL and it creates a CrySL-encoded rule set. Moreover to ease of use CRYSL integrated a Eclipse plugin CogniCrypt [72] and it is called $CogniCrypt_{SAST}$ which is a static analysis tool that gives developers the opportunity to test their Android or Java applications according to CRYSL-encoded rules. CogniCrypt is another developer-friendly tool. It is an Eclipse plugin which supports developers to use cryptographic APIs correctly in their Java projects. It has two main features. Firstly, it can generate code which is secure cryptographically and the second one is CogniCrypt can detect cryptographic APIs misuses in the code by using static analysis in the background. There is another program to detect bugs in Java programs called Spotbugst [35]. It has a graphical user interface(GUI) to facilitate the usage. Moreover, it can be used with Eclipse IDE and Gradle as a plugin. Lastly, Coverity [9] is a static code analysis tool. It can detect bugs in JavaC/C++, C#, JavaScript, Ruby, or Python.

Cryptographic keys are the most crucial thing in cryptography since the security of a system depends on it. K-Hunt [75] is a dynamic analysis tool to detect vulnerable cryptographic keys in executable files. First it finds the locations of cryptographic operations, then it detects insecure keys such as keys generated deterministically, keys negotiated insecurely and keys which are recoverable by tracking how the cryptographic keys are generated and propagated.

Furthermore, Chatzikonstantinou et al. [59] analyze 49 Android applications statically and dynamically to detect cryptographic misuses without a tool.

7.2 Detection of cryptographic misuses on iOS platform

In the iOS platform, there exists few works about cryptographic misuses since it is a closed system and hard to analyze. Egele et al. [62] developed a system whose name is PiOS. It builds CFG from decrypted iOS binary and analyzes CFGs to detect the leak of the private data from user to third party.

7.3 Other Related Works

Applications use SSL/TLS protocols to transmit sensitive data. But these protocols can be weak against the MITM attacks, if they are not implemented well. To investigate misuses related to SSL/TLS protocols in Android platform MalloDroid [64] is developed. It analyzes the static code of an Android application and shows the vulnerabilities. Developers sometimes use custom implementations and it makes Android applications vulnerable to SSL/TLS MITM attacks. SMV-Hunter [88] can check the custom implementations also. Moreover, SMV-Hunter is a hybrid system, that is it uses both static and dynamic analysis in Android applications to detect SSL/TLS vulnerabilities. In the static analysis phase, possible vulnerabilities are identified and the information is found to guide dynamic analysis. Then the vulnerable code is triggered under MITM attack. AndroSSL [67] is another tool which evaluates the SSL/TLS connections in Android applications. By using AndroSSL, developers can analyze their application in terms of MITM attacks.

Developers of the detection tools need a dataset which contains cryptographic misuses examples to evaluate their application. For instance, Afrose et al. [49] developed the first comprehensive benchmark which contains 16 types of cryptographic misuses. This bench-mark contains misuses about the hardcoded secrets, asymmetric and symmetric cryptographic primitives, insecure PRNG, insecure hash functions and improper SSL/TLS certificate and hostname verification. It analyzes 4 tools, SpotBugs [36], Coverity [9], CryptoGuard [84] and CrySL [71].

Wickert et al. [93] provides a dataset including the 201 cryptographic API misuses which gathered from GitHub Projects. Moreover, they integrated the dataset to MUBench [52, 25].

Up to now, as we know there is no study which gathers all up to date tools on various platforms with their methodology. Moreover we give a comprehensive definition of the cryptographic rules.

CHAPTER 8

CONCLUSION

Detection of cryptographic misuse is a new and emerging field. There are many tools developed today. To the best of our knowledge, it is the first study which gathers nearly all cryptographic misuse detection tools. In this work, we studied 11 cryptographic misuse detection tools. They are CryptoLint [61], CMA [87], CDRep [77], sPECTRA [68], BinSight [79], CHIRON [85], CryptoGuard [84] and CRYLOGGER [83] in Android platform; iCryptoTracer [76] and Automated Binary Analysis [65] on iOS platform and CRYPTOREX [94] on IoT platform. We give information about their methodology and the rule set each of them is used by the tool. As a contribution, we show the comprehensive rule set which can be detected tools and also the shortcomings of the tools. In the light of this information, new tools can be developed. This study can be a source for the features and methodology of a new tool to be developed. We believe that our work will be a guide for developers who want to test their application in terms of cryptographic security also. Using our study, comprehensive information can be obtained about the tools developed in this field, the methodologies used by these tools and the misuses they can detect. As our future work, we plan to add new rules to our rule set that could not be detected by detection tools before, and to reveal an expanded rule set.

REFERENCES

[1] Androguard Documentation, https://androguard.readthedocs.io/en/latest/, accessed: 2021-08-15.

[2] Androguard Documentation, https://code.google.com/p/androguard/, accessed: 2021-08-15.

[3] Angr, https://github.com/angr/angr, accessed: 2021-08-15.

[4] APIMonitor, https://code.google.com/archive/p/droidbox/wikis/APIMonitor.wiki, accessed: 2021-08-15.

[5] Apktool, https://ibotpeaches.github.io/Apktool/, accessed: 2021-08-15.

[6] Apktool Installation, https://ibotpeaches.github.io/Apktool/install/, accessed: 2021-08-15.

[7] Attacking ECB, https://zachgrace.com/posts/attacking-ecb/, accessed: 2021-08-15.

[8] Binwalk, https://github.com/ReFirmLabs/binwalk, accessed: 2021-08-15.

[9] Coverity, https://scan.coverity.com/, accessed: 2021-08-15.

[10] Crylogger Source Code, https://github.com/lucapiccolboni/crylogger, accessed: 2021-08-15.

[11] Cryptographic Key Length Recommendation, https://www.keylength.com/en/4/, accessed: 2021-08-15.

[12] CWE Version 4.5, https://cwe.mitre.org/data/published/cwe_latest.pdf, accessed: 2021-08-15.

[13] Generation of Weak Initialization Vector (IV), https://cwe.mitre.org/data/definitions/1204.html, accessed: 2021-08-15.

[14] Google Play Store, https://play.google.com/store, accessed: 2021-08-15.

[15] IDA Pro, https://hex-rays.com/ida-pro/, accessed: 2021-08-15.

[16] Improper Certificate Validation, https://cwe.mitre.org/data/definitions/295.html, accessed: 2021-08-15.

[17] Improper Check for Certificate Revocation, https://cwe.mitre.org/data/definitions/299.html, accessed: 2021-08-15.

[18] Improper Following of a Certificate's Chain of Trust, https://cwe.mitre.org/data/definitions/296.html, accessed: 2021-08-15.

[19] Improper Validation of Certificate Expiration, https://cwe.mitre.org/data/definitions/298.html, accessed: 2021-08-15.

[20] Improper Validation of Certificate with Host Mismatch, https://cwe.mitre.org/data/definitions/295.html, accessed: 2021-08-15.

[21] Inadequate Encryption Strength, https://cwe.mitre.org/data/definitions/326.html, accessed: 2021-08-15.

[22] Key Exchange without Entity Authentication, https://cwe.mitre.org/data/definitions/322.html, accessed: 2021-08-15.

[23] Monkey Documentation, https://developer.android.com/studio/test/monkey, accessed: 2021-08-15.

[24] monkeyrunner Documentation, https://developer.android.com/studio/test/monkeyrunner, accessed: 2021-08-15.

[25] MUBench Source, https://github.com/stg-tud/MUBench, accessed: 2021-08-15.

[26] NIST Bad Password, https://github.com/cry/nbp, accessed: 2021-08-15.

[27] Not Using an Unpredictable IV with CBC Mode, https://cwe.mitre.org/data/definitions/329.html, accessed: 2021-08-15.

[28] Reusing a Nonce, Key Pair in Encryption, https://cwe.mitre.org/data/definitions/323.html, accessed: 2021-08-15.

[29] Reversible One-Way Hash, https://cwe.mitre.org/data/definitions/328.html, accessed: 2021-08-15.

[30] SecLists, https://github.com/danielmiessler/SecLists, accessed: 2021-08-15.

[31] SecureRandom function usage, https://developer.android.com/reference/java/security/SecureRandom, accessed: 2021-08-15.

[32] Security Best Practices: Symmetric Encryption with AES in Java and Android, https://proandroiddev.com/security-best-practices-symmetric-encryption-with-aes\-in-java-7616beaaade9, accessed: 2021-08-15.

[33] Special Publication 800-57 Part 1Revision 5 Recommendation for Key Management: Part 1 – General, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf, accessed: 2021-08-15.

[34] sPECTRA source code, https://bitbucket.org/spectra2016/sourcecode/src/32021e83cec2, accessed: 2021-08-15.

[35] SpotBugs Documentation Release 4.2.2, https://spotbugs.readthedocs.io/en/stable/, accessed: 2021-08-15.

[36] SpotBugs Documentation Release 4.2.2, https://spotbugs.readthedocs.io/en/stable/, [Online; accessed 15-August-2021].

[37] Stack Overflow, https://stackoverflow.com/, accessed: 2021-08-15.

[38] Storing Passwords in a Recoverable Format, https://cwe.mitre.org/data/definitions/257.html, accessed: 2021-08-15.

[39] Use of a Broken or Risky Cryptographic Algorithm, https://cwe.mitre.org/data/definitions/327.html, accessed: 2021-08-15.

[40] Use of a Key Past its Expiration Date, https://cwe.mitre.org/data/definitions/324.html, accessed: 2021-08-15.

[41] Use of a One-Way Hash with a Predictable Salt, https://cwe.mitre.org/data/definitions/760.html, accessed: 2021-08-15.

[42] Use of a One-Way Hash without a Salt, https://cwe.mitre.org/data/definitions/759.html, accessed: 2021-08-15.

[43] Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG), https://cwe.mitre.org/data/definitions/338.html, accessed: 2021-08-15.

[44] Use of Hard-coded Cryptographic Key, https://cwe.mitre.org/data/definitions/321.html, accessed: 2021-08-15.

[45] Use of Password Hash With Insufficient Computational Effort, https://cwe.mitre.org/data/definitions/916.html, accessed: 2021-08-15.

[46] Use of RSA Algorithm without OAEP, https://cwe.mitre.org/data/definitions/780.html, accessed: 2021-08-15.

[47] Veracode state of software security 2017, https://www.veracode.com/sites/default/files/pdf/resources/reports/report-state-of-software-security-2017-veracode-report.pdf, accessed: 2021-08-15.

[48] Weak Password Requirements, https://cwe.mitre.org/data/definitions/521.html, accessed: 2021-08-15.

[49] S. Afrose, S. Rahaman, and D. Yao, CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses, pp. 49–61, 09 2019.

[50] N. J. Al Fardan and K. G. Paterson, Lucky Thirteen: Breaking the TLS and DTLS Record Protocols, pp. 526–540, 2013.

[51] N. AlFardan, D. Bernstein, K. Paterson, B. Poettering, and J. Schuldt, On the security of RC4 in TLS, pp. 305–320, 08 2013.

[52] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, MUBench: A Benchmark for API-Misuse Detectors, p. 464–467, 2016.

[53] E. B. Barker and A. Roginsky, SP 800-131A. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths, 2011.

[54] M. Bellare and P. Rogaway, Optimal Asymmetric Encryption, 950, pp. 92–111, 1994.

[55] D. Bleichenbacher, Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS1, 02 2002.

[56] D. Boneh, A. Joux, and P. Nguyen, Why Textbook ElGamal and RSA Encryption Are Insecure, 1976, pp. 30–43, 10 2000.

[57] A. Braga and R. Dahab, A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software, 11 2015.

[58] D. R. L. Brown, What Hashes Make RSA-OAEP Secure?, 2006, dbrown@certicom.com 13733 received 30 Jun 2006, last revised 8 Aug 2007.

[59] A. Chatzikonstantinou, M. Group, C. Ntantogian, C. Xenakis, and G. Karopoulos, Evaluation of Cryptography Usage in Android Applications, 12 2015.

[60] M. Dworkin, Recommendation for Block Cipher Modes of Operation. Methods and Techniques, p. 67, 12 2001.

[61] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, An empirical study of cryptographic misuse in Android applications, Proceedings of the ACM Conference on Computer and Communications Security, pp. 73–84, 11 2013.

[62] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, PiOS: Detecting Privacy Leaks in iOS Applications, 01 2011.

[63] A. Einarsson and J. Nielsen, A Survivor's Guide to Java Program Analysis with Soot, 08 2021.

[64] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, Why Eve and Mallory love Android: An analysis of Android SSL (in)security, 10 2012.

[65] J. Feichtner, D. Missmann, and R. Spreitzer, Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications, pp. 236–247, 06 2018.

[66] S. Fluhrer, I. Mantin, and A. Shamir, Weakness in the Key Scheduling Algorithm of RC4, Weaknesses in the Key Scheduling Algorithm of RC4, 2259, 08 2001.

[67] F. Gagnon, M.-A. Ferland, M.-A. Fortier, S. Desloges, J. Ouellet, and C. Boileau, AndroSSL: A Platform to Test Android Applications Connection Security, 10 2015.

[68] J. Gajrani, M. Tripathi, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan, sPECTRA: A precise framEwork for analyzing CrypTographic vulneRabilities in Android apps, pp. 854–860, 2017.

[69] B. Kaliski, PKCS #1: RSA Encryption Version 1.5, RFC, 2313, pp. 1–19, 1998.

[70] J. Kelsey, B. Schneier, and D. Wagner, Related-key cryptanalysis of 3-way, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA, Proc First Intl Information and Communication Security, 1334, 05 2000.

[71] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs, 109, pp. 10:1–10:27, 2018, ISSN 1868-8969.

[72] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, CogniCrypt: Supporting developers in using cryptography, pp. 931–936, 2017.

[73] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, Why does cryptographic software fail? A case study and open problems, Proceedings of 5th Asia-Pacific Workshop on Systems, APSYS 2014, 06 2014.

[74] G. Leurent, MD4 is Not One-Way, 5086, pp. 412–428, 02 2008.

[75] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu, K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces, pp. 412–425, 10 2018.

[76] Y. Li, Y. Zhang, J. Li, and D. Gu, iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications, pp. 349–362, 10 2014.

[77] S. Ma, D. Lo, T. Li, and R. Deng, CDRep: Automatic Repair of Cryptographic Misuses in Android Applications, pp. 711–722, 05 2016.

[78] K. Moriarty, B. Kaliski, and A. Rusch, PKCS #5: Password-Based Cryptography Specification Version 2.1, RFC 8018, January 2017.

[79] I. Muslukhov, Y. Boshmaf, and K. Beznosov, Source Attribution of Cryptographic API Misuse in Android Applications, p. 133–146, 2018.

[80] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, Jumping Through Hoops: Why do Java Developers Struggle With Cryptography APIs?, p. 57, 2017.

[81] N. Nethercote and J. Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation, Sigplan Notices - SIGPLAN, 42, pp. 89–100, 06 2007.

[82] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, A Stitch in Time: Supporting Android Developers in WritingSecure Code, pp. 1065–1077, 10 2017.

[83] L. Piccolboni, G. Di Guglielmo, L. Carloni, and S. Sethumadhavan, CRYLOGGER: Detecting Crypto Misuses Dynamically, 07 2020.

[84] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. Yao, CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects, pp. 2455–2472, 11 2019.

[85] S. Rahaman, Y. Xiao, K. Tian, F. Shaon, M. Kantarcioglu, and D. Yao, CHIRON: Deployment-quality Detection of Java Cryptographic Vulnerabilities, 06 2018.

[86] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo, and L. Bassham, NIST Special Publication 800-22: A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications, NIST Special Publication 800-22, 04 2010.

[87] S. Shao, G. Dong, T. Guo, T. Yang, and C. Shi, Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications, 2014.

[88] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps, 2014.

[89] S. Vaudenay, Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS, 2002.

[90] X. Wang and H. Yu, How to Break MD5 and Other Hash Functions, Lecture Notes in Computer Science, 3494, pp. 561–561, 05 2005.

[91] M. Weiser, Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, 1979.

[92] D. L. Wheeler, zxcvbn: Low-Budget Password Strength Estimation, pp. 157–173, August 2016.

[93] A.-K. Wickert, M. Reif, M. Eichberg, A. Dodhy, and M. Mezini, A Dataset of Parametric Cryptographic Misuses, pp. 96–100, 2019.

[94] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, Cryptorex: Large-scale analysis of cryptographic misuse in iot devices, pp. 151–164, September 2019.