CDRep: Automatic Repair of Cryptographic Misuses in Android Applications

Siqi Ma[§], David Lo[§], Teng Li[†], and Robert H. Deng[§]
[§]Singapore Management University, Singapore
[†]Xidian University, Xi'an, China
siqi.ma.2013@phdis.smu.edu.sg
{davidlo, robertdeng}@smu.edu.sg
litengxidian@gmail.com

ABSTRACT

Cryptography is increasingly being used in mobile applications to provide various security services; from user authentication, data privacy, to secure communications. However, there are plenty of mistakes that developers could accidentally make when using cryptography in their mobile apps and such mistakes can lead to a false sense of security. Recent research efforts indeed show that a significant portion of mobile apps in both Android and iOS platforms misused cryptographic APIs. In this paper, we present CDRep, a tool for automatically repairing cryptographic misuse defects in Android apps. We classify such defects into seven types and manually assemble the corresponding fix patterns based on the best practices in cryptographic implementations. CDRep consists of two phases, a detection phase which identifies defect locations in a mobile app and a repair phase which repairs the vulnerable app automatically. In our validation, CDRep is able to successfully repair 94.5% of 1,262 vulnerable apps. Furthermore, CDRep is lightweight, the average runtime to generate a patch is merely 19.3 seconds and the size of a repaired app increases by only 0.667% on average.

Keywords

Cryptographic Misuse, Vulnerability Detection, Automated Program Repair

1. INTRODUCTION

In a remarkably short time, mobile computing has become a fundamental feature in the lives of billions of people, heralding an unprecedented reliance on smart phones and tablets compared to any previous computing technology. With the trend of Bring Your Own Device (BYOD), mobile devices are increasingly used to access and store sensitive corporate information. Malicious attacks to mobile devices and applications therefore not only present a unique set of risks to personal privacy, but also pose new security challenges to enterprise information systems. Many app developers today use cryptographic primitives, such as symmetric key encryption and message authentication codes (MACs), to provide various security services, from user authentication, data privacy, to secure communications. However, developers can easily make mistakes in implementing and using cryptography in their mobile applications due to either a lack of cryptographic knowledge or human error, and such mistakes often lead to a false sense of security.

There are a few efforts in the literature investigating the problem of cryptographic misuses in mobile apps. Egele et al. [11] examined if developers use cryptographic APIs in a fashion that provides typical cryptographic notions of security, For example, indistinguishability under chosen plaintext attack (IND-CPA) security and cracking resistance. They found that about 90% of the 12,000 applications in the Google Play marketplace that use cryptographic APIs make at least one mistake. Shuai et al. [29] built a collection of cryptography misuse models, and implemented an automatic misuse detection tool, Crypto Misuse Analyzer (CMA). They found that more than half of the apps they examined suffer from cryptographic misuses. Li et al. [23] designed a tool called iCryptoTracer which traces cryptographic usage in iOS apps, extracts the trace log and judges whether apps have used cryptography correctly. Veracode in 2013 detected cryptographic usage problems in the source codes of mobile apps and concluded that such problems affect 64% of Android apps and 58% of iOS apps [5]. Given the significant portion of mobile apps affected by cryptographic misuses, it is imperative that such misuses be rectified as soon as possible to avert any potential attacks. Unfortunately, it may not be realistic to expect developers who misused cryptography in the first place to do a good job in fixing the misuses because of their lack of cryptographic knowledge or that they are simply unaware of the problem.

We aim to repair cryptographic misuses in Android apps automatically. There exist a few efforts in automatically repairing software code in the literature. Most of the previous works have focused on fixing general bugs, such as repairing null pointer dereferences. Goues et al. [21] introduced an approach to repair software programs using genetic programming. Kim et al. [18] proposed a novel patch generation approach by first learning common fix patterns from human-written patches and then generating program patches from these common fix patterns. To our knowledge, very few efforts focus on automatic repair of mobile app vulnerabili-

ASIA CCS'16, May 30–June 3, 2016, Xi'an, China.
© 2016 ACM. ISBN 978-1-4503-4233-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897896

ties. Recently, Zhang et al. [33] proposed a technique which generates a patch for component hijacking vulnerability in Android apps; they performed static analysis on bytecode to locate vulnerabilities, and then inserted new code to taint data and tracked and blocked dangerous information flow during runtime. Different from Goues et al. and Kim et al., we focus on specific bugs that correspond to cryptographic misuses. Their generic approaches have low success rates (e.g., only 27 out of 119 bugs are successfully fixed by Kim et al.'s approach). In this work, we make use of specialized domain knowledge to fix specialized bugs to achieve a high success rate. Zhang et al.'s approach can also fix specialized bugs with a high success rate, however, they focus on a different kind of vulnerability with a different analysis method, namely, taint analysis, which is a dynamic analysis. In their work, they used the tainted data to identify the vulnerable sink by keeping track of the taint propagation at runtime. Unfortunately, it is not a effective technique for fixing cryptographic misuses that are considered in this work.

In this paper, we propose CDRep (Cryptographic-Misuse Detection and Repair) which automatically detects and repairs misuses of cryptographic APIs. We focus on the bytecode level patch following the principle provided by Zhang et al. [33], no source code. Thus, we enable to protect users who only have access to the bytecode but not source code of apps. CDRep is designed to repair seven types of misuses identified in [11][29] and operates in two phases: *detection phase* and *repair phase*. In the detection phase, CDRep locates misuses and classifies them following the light-weight static analysis approach proposed by Egele et al. [11]. In the repair phase, CDRep automatically applies and adapts a set of manually created patch templates to repair a vulnerable program. These patch templates can be created one time and used to repair many vulnerable apps with cryptographic misuses. We apply CDRep on 8,640 real-world Android apps and it detects that 8,582 apps have cryptographic misuses. We manually check a random sample of 1,262 apps from the 8,582 apps and among the 1,262 vulnerable apps, CDRep successfully repairs 1,193 of them.

Overall, this paper makes the following contributions:

- We propose and implement CDRep to automatically generate patches to fix cryptographic misuses in Android apps. This is the first effort to repair cryptographic misuses automatically.

- We apply CDRep to 8,640 real-world Android apps. We ask members of our security research team to evaluate the correctness of the automatic repair. Moreover, we email the repaired apps to their developers to check whether CDRep inadvertently changes behaviors of repaired the apps. Our experimental results show that CDRep is able to repair cryptographic misuses effectively, achieving a successful repair rate of 94.5%. A total of 230 developers responded to our emails and 87.0% of them accepted our patches.

1.1 Applications of CDRep

Indeed, the cryptographic misuses could happen due to two reasons:

- Developer lacks the knowledge of cryptography.

- The Android app is developed by an attacker, which means the app is a malicious one.

In view of the above reasons, the cryptographic misuse vulnerability could not be repaired from the developer's perspective. If the developer lacks the knowledge of cryptography, then it might be unable for developer to repair the cryptography misuses correctly. Further, if the Android developer is an attacker, the developer will definitely leave the vulnerabilities which help the attacker collect users' privacy. These cirumstances explain that we are unable to obtain the application source code, namely, the cryptographic misuse could only be repaired on bytecode level.

Handling the repair by users and maintenance companies. CDRep provides a reliable and easy way to repair the cryptographic misuses. Users and maintenance companies enable to repair the vulnerability without the source code of an application. Moreover, CDRep provides the standard implementations for different cryptographic approach. They do no need to have any knowledge of cryptography.

Processing the repair for a batch of apps. The minimum overhead helps users and maintenance companies to process a batch of apps. CDRep assures the minumum changes of the app and the minmum overhead to process the repair. There exist lots of apps that developers will not upgrade or maintain. However, users might still use them. The maintenance companies could use CDRep to fix the cryptographic misuse of those apps.

1.2 Organization

The rest of this paper is organized as follows. Section 2 introduces the types of misuses that CDRep intends to detect and repair. Section 3 presents the overview of our approach. Section 4 elaborates the repair phase of our approach. Experimental results are shown in Section 5. Section 6 discusses the limitation and future work of our approach. Related work is presented in Section 7. Section 8 concludes the paper.

2. CRYPTOGRAPHIC MISUSES

In this section, we list the seven security rules that are used in our work, and any application that violates any of those rules cannot be secure [11][29].

Based on the precisely defined cryptographic algorithms, seven rules are proposed by [11] [29] as follows:

Rule 1: Do not use electronic codebook (ECB) mode for encryption.

Rule 2: Do not use a constant Initialized Vector (IV) for ciphertext block chaining (CBC) encryption.

Rule 3: Do not use constant secret keys.

Rule 4: Do not use constant salts for password-based encryption (PBE).

Rule 5: Do not use fewer than 1,000 iterations for PBE.

Rule 6: Do not use static seeds to seed *SecureRandom*.

Rule 7: Do not use reversible one-way hash (i.e. MD5 message-digest algorithm).

Encryption schemes are used to protect user privacy, ensuring that attackers are unable to extract even a single bit of plaintext from a ciphertext within a reasonable time bound. *Indistinguishability under a chosen plaintext attack* (IND-CPA) is proposed to formalized this notion, and an encryption scheme could be defined as secure if and only if it is IND-CPA secure [11]. However, some encryption mode or wrong implementations make the encryption scheme become non IND-CPA secure, such as using ECB mode and using constant value. Therefore, seven rules are proposed to keep the encryption scheme secure. Based on the seven rules, we defined seven types of misuse, and each misuse violates one of the rules of security. To identify the misuse in a bytecode, we first examine the instruction that is related to the misuse, called *Indicator Instruction.* Then, we locate the instruction that causes the misuse, called *Root Cause Instruction.* We shows seven types of misuses below and the corresponding example bytecode. The root cause of each example is set in bold.

Misuse 1: Using ECB mode to encrypt. ECB mode is not IND-CPA secure in symmetric encryption scheme. The bytecode below shows such misuse. According to the indicator instruction, `Ljava/crypto/Cipher;→getInstance`, we can identify that register v1 holds the value of encryption type. Therefore, we are able to find that value of v1 is "AES/ECB/PKCS5Padding[1]" based on the root cause, which means that the developer uses ECB mode for encryption. Due to that ECB is the default encryption mode set by Android, the developer also uses ECB mode if they only define "AES" in their code.

```
const-string v1, “AES/ECB/PKCSPadding”
invoke-static {v1}, Ljava/crypto/Cipher;→
 getInstance(Ljava/lang/String;)
 Ljava/crypto/Cipher
```

Misuse 2: Using a constant IV in CBC encryption. In CBC encryption scheme, a constant IV will generate a deterministic, stateless cipher, which is not IND-CPA secure. The bytecode snippet below shows such misuse. Instruction, `Ljava/crypto/spec/IvParameterSpec`, is the indicator instruction of this misuse, and we can conclude that register v7 is the IV parameter. However, v7 receives the value from register v10. Thus, v10 holds the original value of the IV, which is set as constant based on the root cause, "1234567898765432". However, the IV should always be set as random based on the CBC encryption construction.

```
new-instance v7,
 Ljavax/crypto/spec/IvParameterSpec;
const-string v10, “1234567898765432”
invoke-virtual v10, Ljava/lang/String/;→
 getBytes()[B
move-result-object v10
invoke-direct {v7, v10}, Ljava/crypto/spec/
 IvParameterSpec;→<init>([B)V
```

Misuse 3: Using a constant secret key. In a symmetric encryption scheme, if an user uses a secret key with insufficient key length, then the attacker can extract the secret key by using dictionary attack. Moreover, if secret key is constant, it will be extracted by using brute force attack. The sample bytecode with a constant secret key is illustrated as below. According to the indicator instruction, `Ljava/crypto/spec/SecretKeySpec`, we are able to define that register v2 holds the value of secret key, and the value of register v2 is "0x0", which is constant.

```
const/4 v2, 0x0
invoke-virtual v2, Ljava/lang/String/;→
 getBytes()[B
move-result-object v2
const-string/jumbo v4, "AES"
invoke-direct {v3, v2, v4},
 Ljava/crypto/spec/SecretKeySpec;→
 <init>([BLjava/lang/String;)V
```

Misuse 4: Using a constant salt in PBE. According to [11], a randomized salt can make PBE perform better. A constant salt makes the algorithm with salt reduce to an algorithm without salt. According to the indicator instruction, `Ljava/crypto/spec/PBEParameterSpec` in the sample bytecode shown as below, we observe that it uses PBE encryption scheme, and register v2 holds a constant salt value, "0x0".

```
const/4 v2, 0x0
new-instance v3, Ljava/crypto/spec/
 PBEParameterSpec;→<init>([BI)V
const/16 v4, 0x64
invoke-direct {v3, v2, v4},
 Ljava/crypto/spec/PBEParameterSpec;
 →<init>([BI)V
```

Misuse 5: Setting iterations < 1,000 in PBE. Based on the suggestion given by PKCS#5[2], the iteration should be at least 1,000 (i.e. 0x3e8 in hexadecimal). According to the indicator instruction, `Ljava/crypto/spec/PBEParameterSpec` in the sample bytecode of misuse 4, it describes the situation where iteration is set inappropriately. Register v4 holds the hexadecimal value of iteration, "0x64" (i.e. 100 in decimal).

Misuse 6: Using a constant seed to seed *SecureRandom* In [3], they show that seeding *SecureRandom* may be insecure since seeding may cause the instance to return a predictable sequence of numbers. If the same seed is reused, the returned number will become repeatable. The indicator instruction in the code shown as below is `Ljava/security/SecureRandom;→getInstance`. Based on the indicator instruction, we can identify the root cause instruction and conclude that *SecureRandom* is seeded by a constant seed, that is, the value of register p0.

[1] AES is the cryptography algorithm chosen by developers. PKCS5Padding is the padding scheme

[2] PKCS#5: Password-Based Cryptography Standard, http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-5-password-based-cryptography-standard.htm.

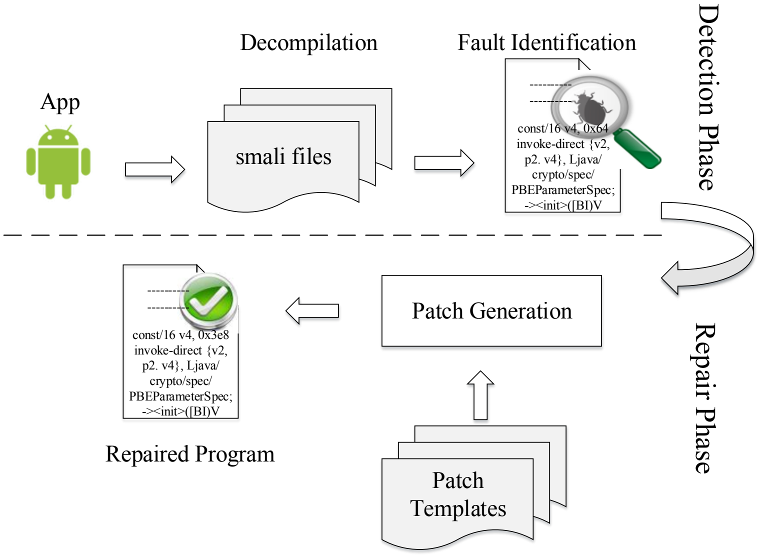


Figure 1: Overview of CDRep

```
p0, [B
...
const-string/jumbo v1, "SHA1PRNG"
invoke-static {v2, v1},
 Ljava/security/SecureRandom;→getInstance
 (Ljava/lang/String;Ljava/lang/String;)
 Ljava/security/SecureRandom;
move-result-object v1
invoke-virtual {v1, p0}, Ljava/security/
 SecureRandom;→setSeed([B)V
```

Misuse 7: Using MD5 hash function. Wang et al. [31] have found many collisions in MD5 and created a powerful attack that can efficiently find MD5 collisions. Based on the indicator instruction, `Ljava/security/MessageDigest` in the sample bytecode below, we infer the root cause instruction and identify that register v2 contains the string of encryption scheme, that is, "MD5", allowing us to conclude that it uses MD5 hash function.

```
const-string v2, "MD5"
invoke-static {v2}, Ljava/security/
 MessageDigest;→getInstance(Ljava/lang/String;)
 Ljava/security/MessageDigest
move-result-object v1
```

3. OVERVIEW OF CDREP

In this section, we introduce the overview of our automatic repair technique, CDRep (Cryptographic-Misuse Detection and Repair). Figure 1 shows the workflow of CDRep. It has two phases, *detection* and *repair*:

Detection: In this phase, CDRep follows the detection steps of CRYPTOLINT [11], which include *decompilation* and *fault identification*. After decompiling an Android app, in the fault identification phase, CDRep checks if vulnerabilities exist in the app; if they exist, CDRep identifies vulnerable Java classes as well as their vulnerability types.

Vulnerabilities are found by first locating indicator instructions (see Section 2) in the decompiled code. Next, for each indicator instruction, CDRep identifies other instructions that the indicator instruction is data dependent on. CDRep then checks all such instructions to identify root causes that correspond to cryptographic misuses. For each misuse, CDRep records its type and the Java class that contains it. Since this step closely follows CRYPTOLINT, we only briefly describe its intuition. Details are available in the original CRYPTOLINT paper [11]. CRYPTOLINT detects six kinds of vulnerabilities (misuses 1-6); in this work, we add one more vulnerability (misuse 7). The procedure to identify the seventh vulnerability is the same as the one used to identify the other six.

Repair: In this phase, CDRep fixes the vulnerable program by performing a series of program transformations specified in a set of manually created patch templates. Details of this phase is presented in Section 4.

4. CRYPTOGRAPHIC MISUSES: AUTOMATIC REPAIR

In this section, we elaborate the repair phase of CDRep. This phase requires a set of manually created patch templates, which we describe in Section 4.1. Given a vulnerable Java class and a vulnerability type, CDRep applies a corresponding patch template to repair the class (described in Section 4.2).

4.1 Patch Templates

We manually create seven patch templates, each for a misuse type. To generate these templates, we take a set of programs with cryptographic misuses and manually fix them. Next, for each pair of correct and faulty program pairs (i.e., with and without cryptographic misuses), we examine code that needs to be added and removed to transform the faulty program to the correct one. We then generalize the added and removed code as a generic patch. A generic patch consists of a series of code *transformations*. Each transformation corresponds to a series of code removal and addition given a particular context. To make the patch generic, we re-

Table 1: Patch overview

Misuse	Patch Overview
1	Using CTR mode.
2	using a randomized IV for CBC encryption.
3	Using a randomized secret key.
4	Using a randomized salt in PBE.
5	Setting iterations = 1,000.
6	Calling *SecureRandom.nextBytes()*.
7	Using SHA-256 hash funtion.

place actual register/variable names, with *placeholders*. We also replace each constant value with a wildcard character ("*") that can match any constant.

Figure 2 presents a sample template for transforming a Java class *Target* containing cryptographic misuse 2, i.e., it uses a constant IV for CBC encryption. The template contains 4 transformations: *i*, *ii*, *iii*, *iv*. Transformation *i* specifies the insertion of the bytecode of java.security.SecureRandom class to the app (if it does not exist). Transformation *ii* specifies the insertion of a field **IvParameterSpec** to *Target*. Transformations *iii* and *iv* specifies code additions (marked by "+") and code deletions (marked by "-") along with a context (marked by "="). Each transformation specifies that whenever a piece of code matches with the context, the lines of code marked by "-" will be replaced with the lines of code marked by "+". In the two transformations, we have placeholders (e.g., $Pl_1, \ldots Pl_6$ in transformation *iii*) and wildcard characters (e.g., "*" at line 1 of transformation *iii*).

It is worth mentioning that cryptographic algorithms always appear in pairs (i.e. encrypt and decrypt). Transformation *iii* is to fix the encryption method in the vulnerable class and transformation *iv* fixes the decryption method. In transformation *iii*, we match for code `Ljava/crypto/spec/IvParameterSpec` to locate indicator instruction (line 5). Then, we replace code that is the root cause of the misuse (line 1) with code that generates the randomized value (line 6-20). The randomized value is stored in the field **ivParams** (line 8). Then we check the length of the randomized value (line 11). Due to that the length is longer than the required length, we only take the sub-length of the randomized value (line 15-18). Finally, the sub-length of randomized value is transformed to the placeholder of IV. Similar to transformation *iii*, we also match the indiator instruction `Ljava/crypto/spec/IvParameterSpec` first in transformation *iv* (line 5). Then, we locate the root cause in line 1 and replace it with codes that are used to generate the randomized value. Instead of generating a randomized value, we extract the value from the field **ivParams** in line 6. We take the same steps as tranformation *iii* to check the length of randomized value and take the required length of randomized value (line 9-16).

Another sample template for transforming a Java class *target* containing cryptographic misuse 5 is shown in Figure 3, i.e., it sets iterations < 1,000. The template only has one transformation, **modification**. However, the modification transformation should be applied on both encryption and decryption method. We first match the code `Ljava/crypto/spec/PBEParameterSpec;`→`<init>` in line 4. Then, we locate the root cause in line 3 and replace it with line 5 to modify the iterations to 1,000.

Due to space constraint, we cannot show all the 7 templates. A brief description of these templates is given in Table 1. We describe another two templates in the Appendix, and the other three in the technical report [4]. Although the generation of these patch templates is manual, it is a one-time cost and the patch templates can be reused to automatically fix many cryptographic misuses.

4.2 Patch Generation

In this step, CDRep takes a vulnerable Java class along with a misuse type as inputs, and generates a patched class. To generate the patched class, CDRep picks the corresponding patch template, runs a series of program transformations specified in the template, and replaces placeholders with actual register/variable names.

Given a transformation, CDRep matches the lines of code marked by "=" and "-" in the vulnerable Java class. In the process, the *mappings* between placeholders and actual variable/register names are identified. The lines of code marked with "-" are then replaced with the lines of code marked with "+". Placeholders in these newly added lines of code are then replaced with actual register names based on the mappings that are identified earlier. Other placeholders in the newly added code that do not appear in the mapping are replaced with new variable/register names that do not appear in the vulnerable Java class.

An automatic code fix example for misuse 2 (i.e., using a constant IV for CBC encryption) is shown in Figure 4. According to the transformation *iii* given in the template, Figure 2, CDRep first matches the line of code marked by "=" (i.e., lines 1, 3, 4, 5 in the vulnerable code of Figure 4(a)). Then, CDRep performs mapping between the placeholders and the actual variable/register shown in Figure 4(b). It is apparently that register v10 is mapped to placeholder Pl_1, which is the root cause. Register v7 is mapped to placeholder Pl_2, which represents the IvParamterSpec. After mapping the actual registers, CDRep replaces the placeholders that are mapped with the actual registers. For those placeholders that are not mapped with any actual registers, we replace them with other available registers (i.e., v1, v2 shown in Figure 4(c)).

5. EVALUATION

In this section, we present the details and results of our experiments that evaluate the performance of CDRep. Our experiments are designed to answer the following questions:

RQ1 (Success rate) How many misuses can CDRep repair successfully?

RQ2 (Runtime) What is the average time needed for CDRep to generate a patch?

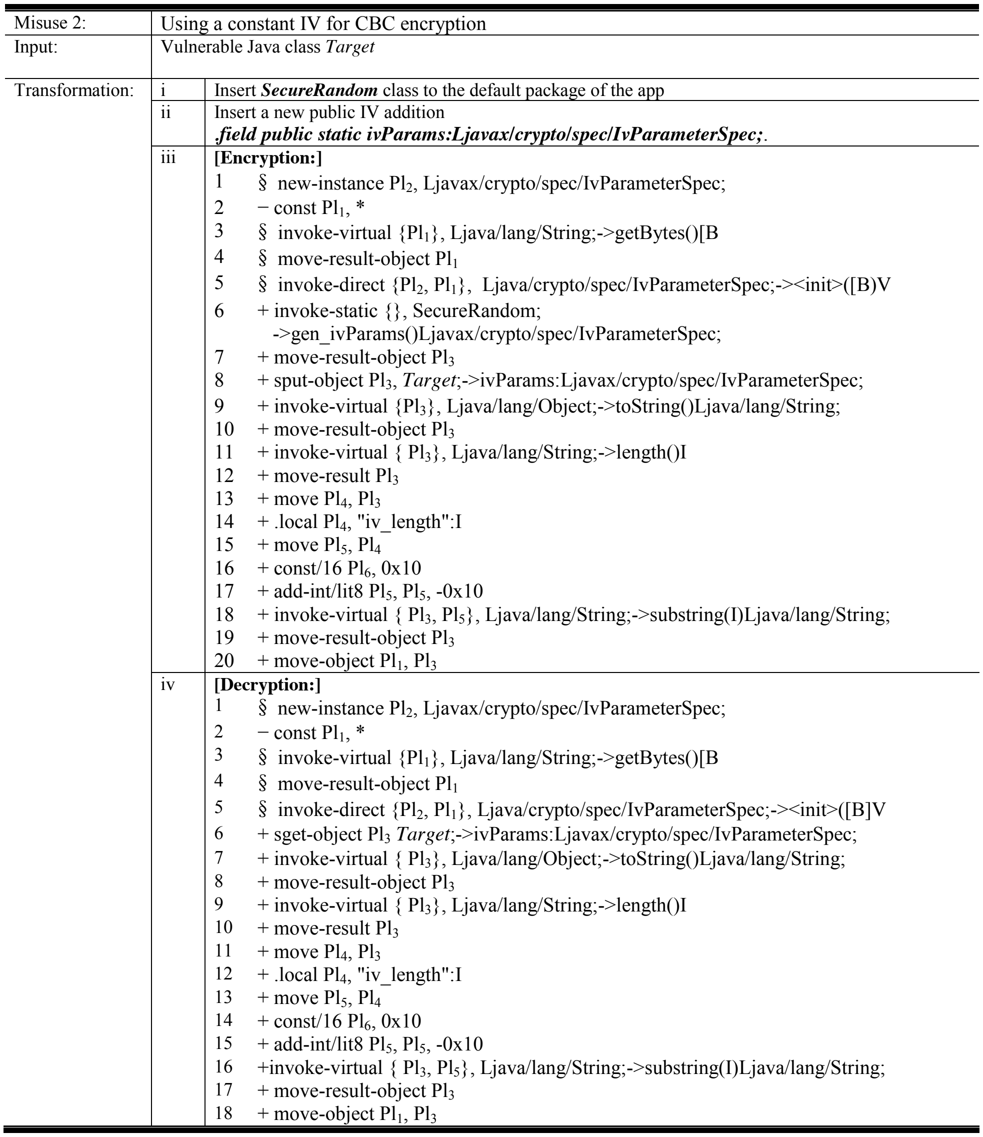
RQ3 (Size) What is the average increase in size of repaired apps?

RQ4 (Failed cases) Why can't some apps be repaired successfully?

5.1 Experiment Setup

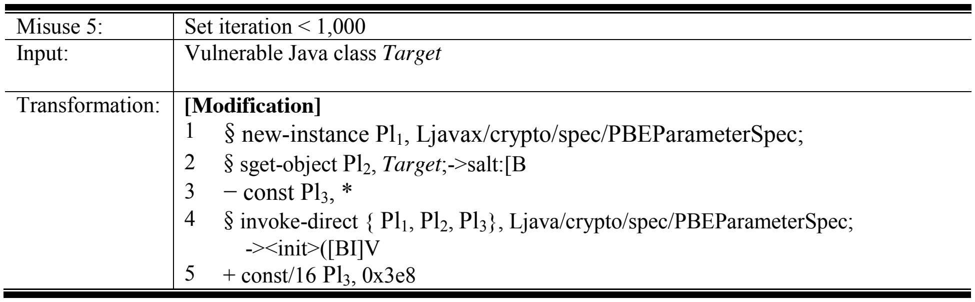
Dataset. To evaluate CDRep, we crawled apps from two app stores, Google play[3] and SlideMe[4] (a third-party store).

[3] Google Play Store: https://play.google.com/store?hl=en
[4] Third-Party store: SlideMe (http://slideme.org/)



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

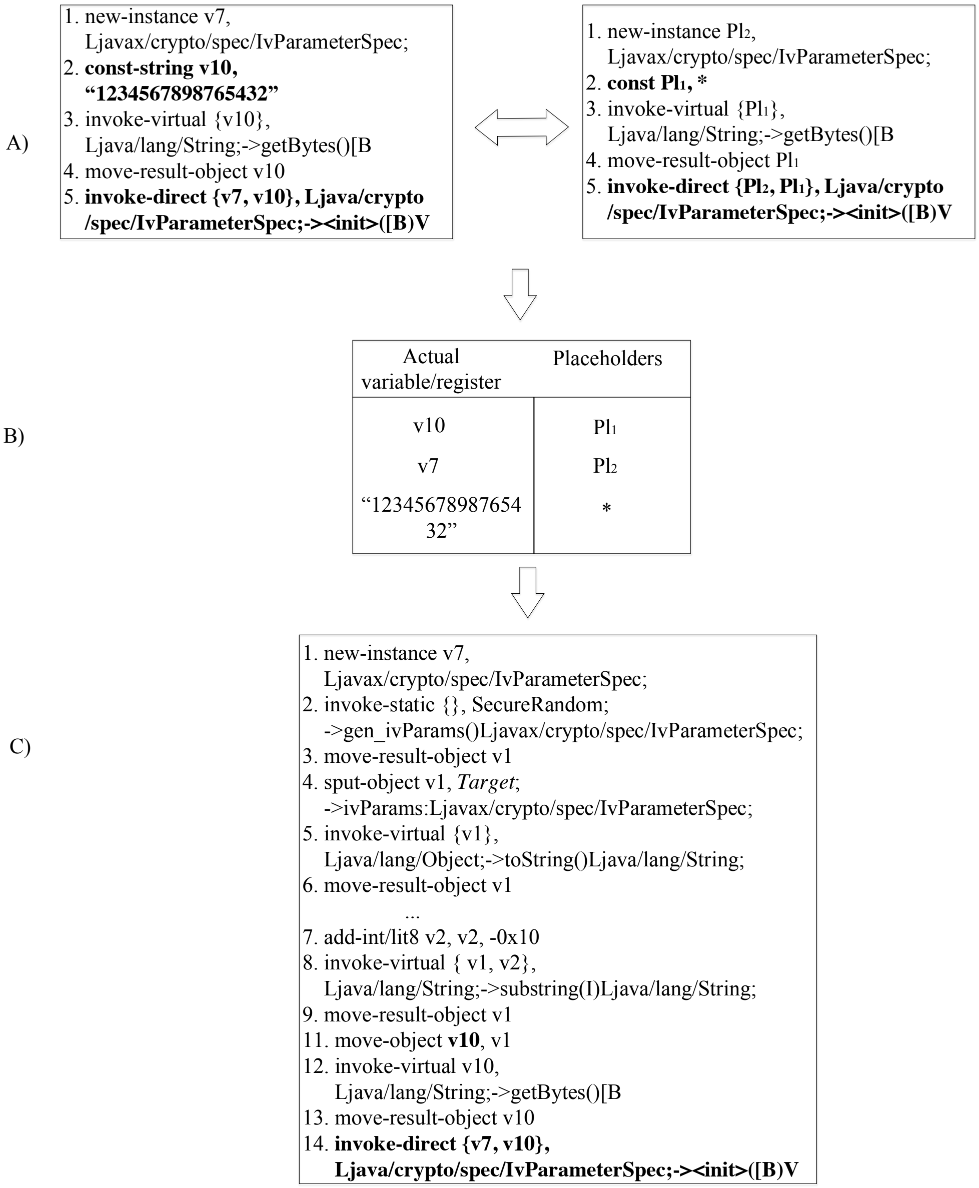
Figure 2: Patch template for misuse 2: this template fixes the misuse that use a constant IV for CBC encryption



Misuse 5:	Set iteration < 1,000
Input:	Vulnerable Java class *Target*
Transformation:	**[Modification]** 1 § new-instance Pl_1, Ljavax/crypto/spec/PBEParameterSpec; 2 § sget-object Pl_2, *Target*;->salt:[B 3 − const Pl_3, * 4 § invoke-direct { Pl_1, Pl_2, Pl_3}, Ljava/crypto/spec/PBEParameterSpec; -><init>([BI)V 5 + const/16 Pl_3, 0x3e8

Figure 3: Patch template for misuse 5: this template fixes the misuse that sets the number of iterations to be less than 1,000

In total, we collected 8,640 free apps (2,114 apps from Google Play, and 6,526 apps from SlideMe), sampled from all categories. Since some sensitive categories (e.g., finance, retail, etc.) are more likely to use cryptography algorithms,



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

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

Actual variable/register	Placeholders
v10	Pl_1
v7	Pl_2
"1234567898765432"	*

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

Figure 4: Fix procedure for misuse 2: it uses a constant IV for CBC encryption. A) shows the vulnerable code with misuse 2, and the template of misuse 2. B) describes the mapping procedure between the actual variable/register extracted from the vulnerable code and placeholders given in the template. C) is the fixed code by replacing the placeholders by the actual registers that are mapped

we sample more applications from these sensitive categories than others (in a ratio of 5.5 to 1).

Detected Cryptographic Misuses. CDRep performs both detection and fix of cryptographic misuses. Table 2 shows the number of cryptographic misuses detected by CDRep across the seven misuse types.

Experiment Design. We evaluate the effectiveness of our approach from three aspects: acceptance rate, patching speed, and size of repaired apps. Acceptance rate evaluates whether our patches are acceptable by security experts and app developers. Patching speed evaluates the efficiency of our approach; if our approach takes a long time to complete, users are less likely to use it. Size of repaired apps is also an important factor that affects usability; if the size of the patched app increases too much, users are less likely to use it.

To measure acceptance rate, we ask our security research team and application developers to examine the repaired programs. Our research team can examine whether the repaired implementations of the cryptographic functionalities are correct. However, they will not be able to conclude whether our patch inadvertently modifies any other behaviours of the app in a bad way. Thus, we also email the repaired apps to their corresponding developers to get feedback on the app behaviours. To measure patching speed, we simply measure the average time that our approach takes to

Table 2: CDRep: Detection result

#	Misuse Type	# of Apps	Percentage	Google Play	SlideMe
1	Use ECB mode	887	10%	402	485
2	Use a constant IV for CBC encryption	979	11%	379	600
3	Use a constant secret key	882	10%	357	525
4	Use a constant salt for PBE	7	0.08%	4	3
5	Set # iterations < 1,000	10	0.1%	7	3
6	Use a constant to seed *SecureRandom*	235	2%	17	218
7	Use MD5 hash function	5582	65%	1359	4223

generate a patch. To measure repaired app size, we measure the percentage of increase in app size after an app has been patched.

To measure acceptance rate, manual inspection (performed by our security research team and app developers) is needed. Since this inspection is a time consuming process, and many apps suffer from cryptographic misuses (see Table 2), it is not possible to check all of the apps that we have repaired (especially for apps that exhibit misuse 1-3, and 7). Thus, except for misuse 4-6 (for which we evaluate all repaired apps), for each other misuse type, we randomly sample apps for manual inspection. For misuse 1, 2, 3, and 7, we select 100, 110, 100, and 700 apps respectively. We vary the number of apps selected for each misuse type, based on the number of apps with cryptographic misuses of that type (we pick around 12% of apps of a particular misuse type).

5.2 RQ1: Success Rate

In this section, we measure how many vulnerable apps are repaired successfully. To make it easier for cryptographers and developers to examine the patch, we not only give them the original vulnerable app and the repaired app that we have repacked, but also provide the bytecode of the vulnerable and repaired apps. In addition, we describe the misuses in the app, and explain why the cryptographic code in the app is not secure.

Table 3 presents the acceptance result of our repaired apps. Overall, our research team accept more than 94.5% of the repaired apps. Considering the email responses, 87% of the repaired program are accepted, which means that the app behaviors are not impacted by the repaired program. According to the result, the patch for misuse 5 and misuse 7 are better than the other types. Our repaired apps are not accepted by all the developers, we explain the reasons in the following section (Section 6).

5.3 RQ2 and RQ3: Runtime and Size

The average runtime needed by our approach to identify a misuse and generate a patch, excluding decompilation time, is only about 19.3 seconds. The bulk of the cost is in the generation of a patch which on average takes 14.6 seconds.

The increase in the size of the patched apps is negligible. Table 4 shows the average increase in the size of patched apps for different misuse types. Across the 7 apps the average increase in size is only 0.667% of the original app size.

5.4 RQ4: Unsuccessful Cases

From Table 3, there are apps that are not repaired successfully by our approach. We discuss the main causes as follows:

Popular libraries. For some apps, developers may call popular libraries. CDRep identifies some misuses that exist in these libraries. For example, several MD5 misuses occur in the classes that are provided by Google, that are, several classes in the "com.google.android.gms.*" package. Although we have repaired those misuses, some app developers rejected our changes since they still prefer to use the standard classes provided by Google.

Incomplete repair: CDRep assumes that each method only contains code that uses one cryptographic scheme. For cases where this assumption does not hold (i.e., a method contains code that uses multiple cryptographic schemes), CDRep could only repair misuses of the first cryptographic scheme. We find that a few apps define more than one encryption scheme in a single method, which causes the patch generated by our approach to be incomplete.

Incomplete decompilation: We use *apktool* to decompile vulnerable apps. However, we find that some apps with complex behaviours cannot be decompiled well. Moreover, some apps reject decompilation. For such cases, CDRep cannot generate patches.

6. LIMITATION

In this section, we discuss the threats to validity. Aside from the limitations corresponding to the unsuccessful cases highlighted in Section 5.4, there are a few other limitations of our approach and its evaluation:

Focus on Android. CDRep is only able to detect and fix cryptographic misuses involving cryptographic classes that come with the Android API. An app may use other third-party cryptographic libraries or implement their own. CDRep is not able to detect and fix cryptographic misuses for such apps. To detect these misuses, there is a need to create new templates. This effort will pay off if the third party cryptographic libraries are used by many Android apps.

Focus on Free Apps. In our experiment, we only evaluate the effectiveness of CDRep on free apps. These apps might not be representative of paid apps. The implementations of paid applications could be different from those of free apps and these differences may impact the effectiveness of our approach. In the future, we plan to expand our study to evaluate the effectiveness of CDRep on paid apps.

Focus on the Interaction. For some apps, they upload user's data to their server instead of keeping it locally. CDRep only ensures that an app could work normally

Table 3: Success Rate

	# of Selected apps	Team Acceptance	# of Developer Response	Developer Acceptance
Misuse 1	100	91(91%)	21	13(61.9%)
Misuse 2	110	92(83.6%)	16	10(62.5%)
Misuse 3	100	83(83%)	23	18(78.2%)
Misuse 4	7	5(71.4%)	3	2(66.7%)
Misuse 5	10	10(100%)	4	4(100%)
Misuse 6	235	212(90.2%)	20	15(75%)
Misuse 7	700	700(100%)	143	138(96.5%)
Total	1262	1193(94.5%)	230	200(87.0%)

Table 4: Average Patch Overhead of different misuse type

	Misuse 1	Misuse 2	Misuse 3	Misuse 4	Misuse 5	Misuse 6	Misuse 7
Overhead	0.749%	0.640%	0.632%	0.742%	0.634%	0.526%	0.748%

if it processes encryption and decryption on the client side. It might break if this app shares the cryptographic paramters with its server, once we modify the cryptographic method the client side.

7. RELATED WORK

In this section, we discuss the previous works that are related to vulnerability detection, malware detection, and automatic program repair.

7.1 Vulnerability Detection

There are many research efforts focusing on different types of vulnerabilities, such as component hijacking vulnerability [27][28][34], cryptographic misuses [11][29][23], and SSL misuses [12][10][19]. Component hijacking vulnerability is related to the threats among the components in Android architecture that implement the access control improperly on external requests and leak the privacy accidentally. CHEX [27] is proposed to detect the component hijacking vulnerability. It identifies the entry points of an app, and splits the app code into subset of code according to the identified entry points. In our paper, we also split the vulnerable code into methods, which makes the repair more effective. The component vulnerability detection of [28] and [34] focus on detecting the intent and permission use behaviour among the components, respectively. Another type of vulnerabilities is SSL misuses. MalloDroid [12] detects the SSL/TLS in the apps that require INTERNET connection. It detection scheme focuses on the certificate, hostname, and SSL protocol checking. As the HTML5-based application becoming popular, code injection on HTML5-based applications [15][16][14] has caught attentions. Approaches are proposed to detect the privacy leakage and permission leakage caused by using Javascript.

Our work is built upon CRYPTOLINT [11], which is able to detect the misuses of cryptographic primitives (misuse 1-6 in our paper). We follow the same steps as CRYPTOLINT to identify the misuses of cryptographic APIs investigated in this work. CMA [29] can identify additional cryptographic misuses that CRYPTOLINT cannot identify (including misuse 7 in our paper). Li et al. proposed iCryptoTracer [23] which performs dynamic and static analysis to detect cryptographic misuses in iOS apps. In this work, we extend the above mentioned studies by proposing an approach that can not only detect but also *fix* cryptographic misuses.

7.2 Malware Detection

Malware detection is also a significant part in the research. Android malware detection approaches are grouped into three categories, dynamic analysis (e.g., [9], [6]), static analysis (e.g., [30], [13]), and hybrid analysis (e.g., [7]). Burguera et al. [9] proposed an approach that could distinguish the malware from the normal applications by analyzing the application behaviors, although they share the same application information (i.e, version, name). Different from previous detection approach, Arora et al. [6] proposed a novel approach that uses the network traffic analysis to detect malwares. Based on the network traffic features collected from chatting, browsing, and mailing, the authors enabled to build a rule-based classifier to detect malwares. Tuvell and Venugopal [9] summarized the search string features identified from compressed executables based on a family of malware. Through the search string features, it enables to identify those malwares that belong to the existing malware family. Feng et al. [13] presented a new approach, Apposcopy, which identifies a type of Android malware that steals user's private information. It identifies the malware features through Inter-Component Call Graph. FlowDroid [7] provides a novel static taint analysis to extract the callbacks and detect the data leakage in Android apps.

7.3 Automatic Program Repair

GenProg [21], PAR [18], and the state-of-the-art, recently proposed HDRepair [20] are three techniques that are able to repair general software bugs automatically. GenProg performs genetic programming to generate patches. PAR generates patches based on prior human-written patches. PAR authors collect a large number of human-written patches and manually generate 10 patch templates. HDRepair *automatically* analyzes bug fix history to infer many graph-based fix patterns, which are then used to guide a genetic programming solution to generate high-quality patches. These studies focus on generic bugs and have low success rates – the best approach (i.e., HDRepair) can only fix 23 out of the 90 bugs in their experiment study. Our approach focuses on a special family of software bugs, i.e., cryptographic misuses. We leverage specialized domain knowledge (i.e., how to fix cryptographic misuses) and achieve a much higher success rate. Furthermore, GenProg requires many hours to generate a patch while our approach just requires less than a minute to generate a patch. Additionally, none of PAR patch templates and the mutation operators used in HDRepair are designed for cryptographic misuses. Moreover, fixes of cryptographic misuses appear rarely in fix history and thus HDRepair, which relies on fix history, would not be able to fix them well.

Our patch templates are created based on the correct setting of the cryptographic primitives. Most of automated repair approaches are based on the static analysis, Azim et al. [8] proposed a self-healing approach by using dynamic analysis and static analysis. They performed dynamic analysis on the apps' behaviours and detect the crashes. Then, they generated static activity transition graph (SATG) model to seal the point that caused the crash . To test the repaired program, they rollbacked to the nearest safe point before the crash happened, and executed the repaired program.

The closest related work is AppSealer [33], which is the first approach that can fix security vulnerabilities in mobile apps. AppSealer is able to fix vulnerabilities that can be leveraged to perform component hijacking attacks. To fix vulnerabilities, it translates Java bytecode into Jimple IR and track the sensitive information on Jimple IR. They places shadow statements to insert new variables and instructions to track the taint data and block dangerous information. Different from AppSealer, we focus on a different kind of vulnerabilities, i.e., cryptography misuses.

8. CONCLUSION & FUTURE WORK

In this paper, we propose a approach, CDRep, to automatically repair vulnerable apps with cryptographic misuses. Given a vulnerable Android app, we first perform static analysis to locate the misuse and identify the misuse type. Then, based on the misuse type, we apply a suitable patch template and adapt it to the vulnerable program by replacing register placeholders in the template with actual register names. Finally, we perform an optimization step to remove dead code. To evaluate CDRep, we crawled 8,640 real-world Android apps and use CDRep to identify cryptographic misuses and repair them. Out of the repaired apps, we randomly pick 1,262 of them for manual inspection (by security experts and app developers). The evaluation results show that CDRep can automatically repair the vulnerable apps effectively – it is able to repair 94.5% of the 1,262 vulnerable apps with an average patch generation time of merely 19.3 seconds.

There are several aspects for future work. CDRep aims to repair the cryptographic misuse by using static analysis at bytecode level. However, detection with static analysis is not complete.

Detect Self-Written Encryption/Decryption class. In the detection phase of CDRep, we detect the cryptographic misuse by using the pre-defined cryptographic APIs that Java provided (e.g., Cipher.getInstance). However, some developers might prefer to call the cryptographic function written by themselves instead of calling the existing cryptographic APIs. CDRep is unable to detect the self-written encryption/decryption class.

Identify Constant Variable. We adopt backward data analysis to identify the constant variable. However, it could only match the constant variable when it is defined in the function. In some circumstances, value of the variable is not set in the function, and it is assigned by the heap during runtime.

We will extend CDRep by applying hybrid analysis (i.e., static analysis and dynamic analysis). Static analysis enables to extract the cryptographic usage from the code level, and dynamic analysis could caputure the code behaviors at runtime. It helps detect the self-written encryption/decryption class and provide a more completed data flow graph.

9. REFERENCES

[1] Androguard. https://github.com/androguard/androguard.

[2] apktool. https://code.google.com/p/android-apktool/.

[3] Securerandom. http://developer.android.com/reference/java/security/SecureRandom.html#insecure_seed.

[4] Seven templates for the corresponding misuses. https://sites.google.com/a/smu.edu.sg/my-work/home.

[5] Status of software security report, vol 5. http://www.veracode.com/resources/state-of-software-security, 2013.

[6] A. Arora, S. Garg, and S. K. Peddoju. Malware detection using network traffic analysis in android based mobile devices. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 66–71. IEEE, 2014.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[8] M. T. Azim, I. Neamtiu, and L. M. Marvel. Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 623–628. ACM, 2014.

[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[10] M. Conti, N. Dragoni, and S. Gottardo. Mithys: Mind the hand you shake-protecting mobile devices from ssl usage vulnerabilities. In *Security and Trust Management*, pages 65–81. Springer, 2013.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. ACM, 2013.

[12] 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. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[13] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[14] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu. Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. 2015.

[15] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[16] X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for html5-based mobile applications in android. In *Proceedings of the 16th Information Security Conference (ISC)*, 2013.

[17] M. Junjin. An approach for sql injection vulnerability detection. In *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on*, pages 1411–1414. IEEE, 2009.

[18] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[19] S. H. Kim, D. Han, and D. H. Lee. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 659–668. ACM, 2013.

[20] X.-B. D. Le, D. Lo, and C. L. Goues. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[21] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[22] J. League. Proper use of java's securerandom. https://www.cigital.com/justice-league-blog/2009/08/14/proper-use-of-javas-securerandom/.

[23] Y. Li, Y. Zhang, J. Li, and D. Gu. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In *Network and System Security*, pages 349–362. Springer, 2014.

[24] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *ACM SIGPLAN Notices*, 38(5):141–154, 2003.

[25] H. Lipmaa, P. Rogaway, and D. Wagner. Ctr-mode encryption. In *First NIST Workshop on Modes of Operation*. Citeseer, 2000.

[26] H. Lipmaa, D. Wagner, and P. Rogaway. Comments to nist concerning aes modes of operation: Ctr-mode encryption. 2000.

[27] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[28] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*. Citeseer, 2013.

[29] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*, pages 75–80. IEEE, 2014.

[30] G. Tuvell and D. Venugopal. Malware detection system and method for compressed data on mobile platforms, 2015. US Patent 9,009,818.

[31] X. Wang and H. Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.

[32] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[33] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

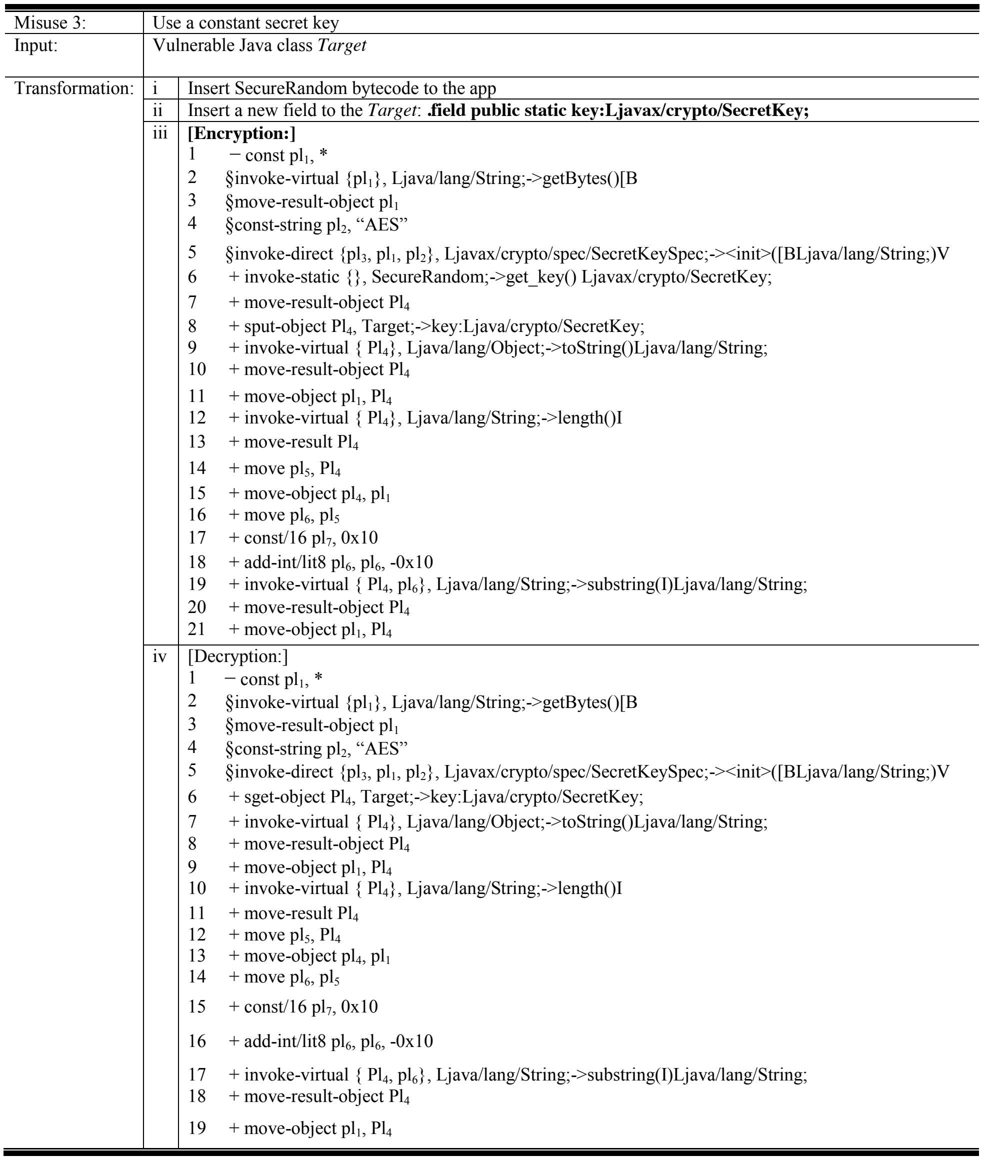
[34] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

Appendix

In this section, we show two other templates for repairing misuse 3 (i.e., use a constant secret key) and misuse 7 (i.e., use MD5 hash function).

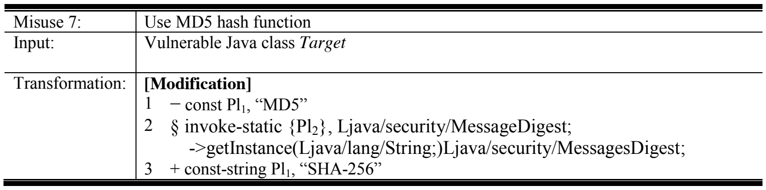
Figure 5 presents the patch template for misuse 3, i.e., use a constant secret key. In the patch template, we create a new class to generate random number. It inserts a new SecureRandom class to generate a new random number in transformation *i*. To store te secret key, we create a new field for the random number in transformation *ii*. In transformation *iii*, we identify the register that stores the original constant secret key, namely, Pl_1, and replace the Pl_1 with the new generated random number. First, the inserted SecureRandom function is called, and it assigns a random number to temporary register Pl_4 from line 6 to line 8. Next, the random string is processed based on the required length from line 10 to line 19. Finally, the processed random string is assigned to the original register Pl_1.

Figure 6 shows the patch template for misuse 7, i.e, use MD5 hash function. Instead of inserting a new class, we first identify the location that calls API `Ljava/security/MessagesDigest`. Based on the function call, we extract the register that keeps the input parameter (i.e., Pl_1). The value stored in Pl_1 corresponds to the encryption method chosen. We modify the value "MD5" into "SHA-256".



Misuse 3:		Use a constant secret key
Input:		Vulnerable Java class *Target*
Transformation:	i	Insert SecureRandom bytecode to the app
	ii	Insert a new field to the *Target*: **.field public static key:Ljavax/crypto/SecretKey;**
	iii	**[Encryption:]**
		1 − const pl_1, *
		2 §invoke-virtual {pl_1}, Ljava/lang/String;->getBytes()[B
		3 §move-result-object pl_1
		4 §const-string pl_2, “AES”
		5 §invoke-direct {pl_3, pl_1, pl_2}, Ljavax/crypto/spec/SecretKeySpec;-><init>([BLjava/lang/String;)V
		6 + invoke-static {}, SecureRandom;->get_key() Ljavax/crypto/SecretKey;
		7 + move-result-object Pl_4
		8 + sput-object Pl_4, Target;->key:Ljava/crypto/SecretKey;
		9 + invoke-virtual { Pl_4}, Ljava/lang/Object;->toString()Ljava/lang/String;
		10 + move-result-object Pl_4
		11 + move-object pl_1, Pl_4
		12 + invoke-virtual { Pl_4}, Ljava/lang/String;->length()I
		13 + move-result Pl_4
		14 + move pl_5, Pl_4
		15 + move-object pl_4, pl_1
		16 + move pl_6, pl_5
		17 + const/16 pl_7, 0x10
		18 + add-int/lit8 pl_6, pl_6, -0x10
		19 + invoke-virtual { Pl_4, pl_6}, Ljava/lang/String;->substring(I)Ljava/lang/String;
		20 + move-result-object Pl_4
		21 + move-object pl_1, Pl_4
	iv	[Decryption:]
		1 − const pl_1, *
		2 §invoke-virtual {pl_1}, Ljava/lang/String;->getBytes()[B
		3 §move-result-object pl_1
		4 §const-string pl_2, “AES”
		5 §invoke-direct {pl_3, pl_1, pl_2}, Ljavax/crypto/spec/SecretKeySpec;-><init>([BLjava/lang/String;)V
		6 + sget-object Pl_4, Target;->key:Ljava/crypto/SecretKey;
		7 + invoke-virtual { Pl_4}, Ljava/lang/Object;->toString()Ljava/lang/String;
		8 + move-result-object Pl_4
		9 + move-object pl_1, Pl_4
		10 + invoke-virtual { Pl_4}, Ljava/lang/String;->length()I
		11 + move-result Pl_4
		12 + move pl_5, Pl_4
		13 + move-object pl_4, pl_1
		14 + move pl_6, pl_5
		15 + const/16 pl_7, 0x10
		16 + add-int/lit8 pl_6, pl_6, -0x10
		17 + invoke-virtual { Pl_4, pl_6}, Ljava/lang/String;->substring(I)Ljava/lang/String;
		18 + move-result-object Pl_4
		19 + move-object pl_1, Pl_4

Figure 5: Patch template for misuse 3: this template fixes the use of a constant secret key



Misuse 7:	Use MD5 hash function
Input:	Vulnerable Java class *Target*
Transformation:	**[Modification]** 1 − const Pl_1, “MD5” 2 § invoke-static {Pl_2}, Ljava/security/MessageDigest; ->getInstance(Ljava/lang/String;)Ljava/security/MessagesDigest; 3 + const-string Pl_1, “SHA-256”

Figure 6: Patch template for misuse 7: this template fixes the use of MD5 instead of a stronger hash function