2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Negative Results on Mining Crypto-API Usage Rules in Android Apps

Jun Gao*, Pingfan Kong*, Li Li†, Tegawendé F. Bissyandé*, Jacques Klein*
*University of Luxembourg, Luxembourg
†Monash University, Australia
{jun.gao, pingfan.kong, tegawende.bissyande, jacques.klein}@uni.lu
li.li@monash.edu

***Abstract*—Android app developers recurrently use crypto-APIs to provide data security to app users. Unfortunately, misuse of APIs only creates an illusion of security and even exposes apps to systematic attacks. It is thus necessary to provide developers with a statically-enforceable list of specifications of crypto-API usage rules. On the one hand, such rules cannot be manually written as the process does not scale to all available APIs. On the other hand, a classical mining approach based on common usage patterns is not relevant in Android, given that a large share of usages include mistakes. In this work, building on the assumption that *"developers update API usage instances to fix misuses"*, we propose to mine a large dataset of updates within about 40 000 real-world app lineages to infer API usage rules. Eventually, our investigations yield negative results on our assumption that API usage updates tend to correct misuses. Actually, it appears that updates that fix misuses may be unintentional: the same misuses patterns are quickly re-introduced by subsequent updates.**

I. Introduction

Although software systems have greatly impacted the efficiency of transactions and communications in our digital world, the security and privacy issues that they carry have been raising concerns among all stakeholders. In this context, the software development community is now urged to implement means to protect user assets, most notably by using cryptography for ensuring confidentiality of data and transactions, as well as the authenticity of information. Unfortunately, several recent studies [1], [2], [3] have revealed that developers often make mistakes when using cryptography APIs (hereafter, crypto-APIs is used for short), even those APIs implemented in widely used libraries such as the Java Cryptography Architecture (JCA). These misuses, which may lead to security mishaps [4], [5], actually carry an illusion of safety for users and developers. Consequently, the research community has started a new effort towards improving the analysis and fix of crypto-APIs usage [6], [7].

To properly use crypto-APIs, developers must learn the API usage rules. Similarly, to validate code, the research and practice communities must build tools for checking API calls against a database of the associated API usage rules. To the best of our knowledge, there are three strategies commonly adopted in the literature for the inference of general API usage rules:

- **manual specification:** Recent literature on static analysis for verifying crypto-API usages propose approaches that are based on manually-written specifications [6], [8]. Although such approaches offer a high degree of reliability, they may require extensive security expertise, and do not scale to the sheer number of cryptography libraries (and their associated recurrent API updates).
- **majority contest of usage patterns:** A trivial approach for systematically finding and updating API protocols is to mine usage patterns in a representative dataset of developer code [9], [10]. Most recurrent patterns are considered as the correct protocol. Such approaches have been shown effective in operating system code [11] where the majority of developers have a significant level of expertise [12], [13]. In the Android community, however, most developers are novice and their usages of crypto-APIs are generally incorrect [14].
- **commit log mining:** Recently, Paletov *et al.* [7] have proposed to mine commit messages from software version tracking systems to identify fixes of API usages and infer the "correct" usages based on static code analysis. Theoretically, this strategy is reliable (in contrast to simple popularity voting of usage patterns). In practice, however, developers often make uninformed updates, and it is now accepted that commit messages are often less informative than what researchers expect [15].

This paper. Our work is set in the context of the Android development community where millions of apps are built and regularly updated on markets. Our objective is to present and investigate the suitability of an approach to infer crypto-API usage rules based on developer updates. Although most of Android apps are not associated with public source code management systems, their different apk releases can be readily reverse-engineered into intermediate representations (e.g., *smali* or *Jimple*) by using frameworks such as *Apktool* and *Soot* [16]. These representations can then be statically analyzed in a straightforward way for extracting usage instances, and comparing usages across updates. Our approach will leverage the AndroZoo [17] dataset where successive apk releases are continuously crawled for the research community.

> *Our main assumption for inferring crypto-API usage rules by mining code updates is that "***API usage updates generally transform incorrect usages into correct usages***". Although this assumption is intuitively reasonable, our investigations have yielded contradictory results. We*

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00065

thus report on the negative results of mining crypto-API usage rules by mining Android app updates. We focus in this study on the widespread JCA APIs used in 598,875 apk releases associated with 39,213 lineages of real world Android apps.

II. Background

We now provide details on the crypto-APIs studied in this work as well as the tool chain leveraged for statically checking API usages (i.e., to build the ground truth for the study).

A. Crypto-APIs

Cryptography is the science that yields algorithms for hiding information from third-parties. Encryption and decryption mechanisms are used to support authentication as well as to guarantee the confidentiality of transactions and information integrity. In software development, crypto-APIs are provided as part of programming toolkits to accelerate the inclusion of cryptography functionalities in developer code. For example, in the Java realm, the Java Cryptography Architecture (JCA) APIs, which are officially provided by Oracle [18], are widely used by developers. Given that most Android apps are built in Java, the use of JCA APIs is also widespread in the Android community. Although some alternate crypto-APIs, such as Apache Commons Crypto [19] APIs, do exist, they are substantially less widespread. Therefore, our work is focused on JCA to investigate the potential API misuses among real-world apps.

Table I enumerates the 23 API classes implemented in the JCA library, where each class was designed to address a specific cryptography functionality: for example, the *MessageDigest* class includes algorithm implementations for computing the digest of some information (e.g., text message) which can be used to check its integrity after transmission.

TABLE I: Java Cryptography Architecture (JCA) APIs.

API Class: *Description*
java.security.AlgorithmParameters: maintainer for security parameters for specific algorithms
javax.crypto.Cipher: provide encryption and decryption functionality
javax.crypto.spec.DHGenParameterSpec: parameters for generating **D**iffie-**H**ellman parameters for DH key agreement
javax.crypto.spec.DHParameterSpec: parameters used for DH algorithm
java.security.spec.DSAGenParameterSpec: parameters for DSA parameter generation
java.security.spec.DSAParameterSpec: parameters used for DSA algorithm
javax.crypto.spec.GCMParameterSpec: parameters for cipher using **G**alois/**C**ounter **M**ode
javax.xml.crypto.dsig.spec.HMACParameterSpec: parameters for the XML signature HMAC algorithm
javax.crypto.spec.IvParameterSpec: Initialization **V**ector for block cipher
javax.crypto.KeyGenerator: generate keys for encryption-decryption
java.security.KeyPair: holder of a publicprivate key pair
java.security.KeyPairGenerator: create publicprivate key pairs
java.security.KeyStore: a memory storage to maintain keys and certificates for later usage
javax.crypto.Mac: Message **A**uthentication **C**ode for message integrity protection
java.security.MessageDigest: a one-way hash for messages
javax.crypto.spec.PBEKeySpec: specification of a **P**assword **B**ased **E**ncryption key
javax.crypto.spec.PBEParameterSpec : parameters for password based encryption
java.security.spec.RSAKeyGenParameterSpec: parameters for RSA key pair generation
javax.crypto.SecretKey: a symmetric secret key
javax.crypto.SecretKeyFactory: convert key into key specification and vice-versa
javax.crypto.spec.SecretKeySpec: specification of a symmetric secret key
java.security.SecureRandom: generate secured pseudo-random numbers
java.security.Signature: digital signature

Implementation-wise, to perform a cryptography-related task, an object associated with the relevant JCA class must first be instantiated. Subsequently, a sequence of the object methods is invoked in a specific order of steps. Listing 1 shows a usage example of API *PBEKeySpec* retrieved from a human resource management app named *com.successfactors.android*. The code snippet is written in *Jimple*, the intermediate representation of *Soot*, which we leveraged in this work to reverse engineering Android apps.

First, the *password* (i.e., $r3) and *salt* (i.e., $r0) parameters of the constructor of *PBEKeySpec* are initialized with the passed-in arguments (lines 7-10). Then, an object of *PBEKeySpec* is constructed with the *password* and *salt* (lines 11-12). There are 2 extra constants of type *int* used when instantiating the object (line 12): the first one, (1 000 in this example), is used to specified the iteration number, the second one (i.e., 256) is used to specify the key length. The *PBEKeySpec* object is used to further generate a *SecretKey* object (line 13-14). After using the *PBEKeySpec* object, for security consideration, the password is cleared from the memory (line 15). The rest part of the example is to create a *SecretKeySpec* by using the previously generated objects and return it for other utilizations.

```
1  private static javax.crypto.spec.SecretKeySpec
       deriveEncryptrionKey(char[], byte[])
2      {
3          javax.crypto.spec.PBEKeySpec $r2;
4          javax.crypto.SecretKeyFactory $r5;
5          javax.crypto.SecretKey $r6;
6          javax.crypto.spec.SecretKeySpec $r7;
7          byte[] $r0;
8          char[] $r3;
9          $r0 := @parameter1: byte[];
10         $r3 := @parameter0: char[];
11         $r2 = new javax.crypto.spec.PBEKeySpec;
12         specialinvoke
       $r2.<javax.crypto.spec.PBEKeySpec: void
       <init>(char[],byte[],int,int)>($r3, $r0, 1000,
       256);
13         $r5 = <com.sybase.persistence.SharedDataVault:
       javax.crypto.SecretKeyFactory secretKeyFactory>;
14         $r6 = virtualinvoke
       $r5.<javax.crypto.SecretKeyFactory:
       javax.crypto.SecretKey
       generateSecret(java.security.spec.KeySpec)>($r2);
15         virtualinvoke
       $r2.<javax.crypto.spec.PBEKeySpec: void
       clearPassword()>();
16         $r7 = new javax.crypto.spec.SecretKeySpec;
17         $r0 = interfaceinvoke
       $r6.<javax.crypto.SecretKey: byte[]
       getEncoded()>();
18         specialinvoke
       $r7.<javax.crypto.spec.SecretKeySpec: void
       <init>(byte[],java.lang.String)>($r0, "AES");
19         return $r7;
20     }
```

Listing 1: JCA API Usage Example (Jimple code representation)

In this example, there are several code locations where developers can make mistakes that would lead to misuses of the *PBEKeySpec* API:

- (line 9) - Security strength of a password is heightened when *salt* is properly generated in a random way. In practice, however, developers commonly hard code their *salt* value. In the example code, *salt* is specified as parameter of method *deriveEncryptrionKey* (line 1) and then stored in

$r0. So, a misuse could happen if a constant is passed to *deriveEncryptrionKey* in the second parameter.

- (line 10) - Often, developers use a *String* object to hold the password and then use *toCharArray()* to convert to the required type (i.e., *char[]*) when necessary. However, the intention of designing *PBEKeySpec* constructor to only accept *char[]* instead of *String* is to avoid using *String*, since *String* object is immutable, therefore, they cannot be destroyed or modified after instantiation until *garbage collection* revokes the memory. Given that *garbage collection* occurs randomly and is out of the control of developers, the password can survive in memory for a long time, increasing the risk of being exploited by attacks.
- (line 12) - Documentation of JCA recommends an iteration number above 1 000. It is however common to have cases where developers, with little expertise, assign a smaller iteration rate.
- (line 15) - Password information should be kept in memory only for the duration it is needed, in order to minimize attack opportunities. Thus it should not be held in a *String* object and must be cleared immediately after the use of the *PBEKeySpec* object. Developers unfortunately often overlook the call to the *clearPassword()* of *PBEKeySpec*. In this example code, the method *clearPassword* in line 15 is correctly called, so there is no misuse.

```
1  Findings in Java Class: com.umeng.common.util.h
2
3   in Method: java.lang.String a(java.lang.String)
4      ConstraintError violating CrySL rule for MessageDigest
5          First parameter (with value "MD5") should be any of {SHA-256, SHA-384,
           SHA-512}
6              at statement: $r2 = staticinvoke <java.security.MessageDigest:
           java.security.MessageDigest getInstance(java.lang.String)>("MD5")
7
8      TypestateError violating CrySL rule for MessageDigest
9          Unexpected call to method reset on object of type
           java.security.MessageDigest. Expect a call to one of the following
           methods digest,update
10             at statement: virtualinvoke $r2.<java.security.MessageDigest: void
           reset()>()
```

Listing 2: CogniCrypt_SAST Report Example

B. Static API usage checker

We leverage $CogniCrypt_{SAST}$ [20], a static analyzer of the *CogniCrypt* [21] framework, for detecting JCA API misuses in Java programs. This analyzer was selected as it has been extended to be compatible with Android apps as well [20], a static analyzer of the [22]. $CogniCrypt_{SAST}$ checks JCA APIs against a set of rules that were manually specified by security experts using *CrySL* [6] (CogniCrypt Specification Language). We consider CrySL rules as a reliable and accurate oracle for deciding whether a JCA API is misused or not. Concretely, given an Android apk, $CogniCrypt_{SAST}$ identifies all instances of the 23 JCA API classes and checks the usage against the CrySL rules to generate a report on all detected misuses. Listing 2 showcases an example of a report generated by $CogniCrypt_{SAST}$ where misuses are hierarchically grouped by classes and methods. In this example, 2 misuses are found in method *java.lang.String* of class *com.umeng.common.util.h* of app *com.lovinc.radio*:

- The first misuse, a *ConstraintError* of API *Messagedigest*, is reported in line 4, with details indicating that argument *"MD5"* is not recommended when invoking API method *getInstance(java.lang.String)*.
- The second misuse (*TypestateError* error) also relates to the same API object *$r2* and occurs in the same method. It specifies that the misuse is caused by the fact that the *MessageDigest* object had not been in the state to call method *reset()*, instead, method *digest* or *update* should be invoked before.

As demonstrated by this example, a single JCA API usage instance can suffer from multiple misuse errors. Table II enumerates and provides brief explanations on 6 misuse types detected by $CogniCrypt_{SAST}$. Actually $CogniCrypt_{SAST}$'s reports may include *ImpreciseValueExtractionError* notifications, which indicate that $CogniCrypt_{SAST}$ cannot obtain all the information for the analysis, and thus no clear conclusion can be given. We do not discuss such cases in this study. Nevertheless, given that we must be able to assess whether a misuse has actually been fixed in an app update, we must be able to enumerate all usage locations. To that end, we have developed on top of *Soot* [16] a dedicated tool for supporting the extraction of JCA API usage instances.

TABLE II: CogniCrypt Misuse Types.

Type	**Explanation**
Example	
ConstraintError	Unrecommended arguments are given.
e.g., *MD5* as hashing algorithm.	
RequiredPredicateError	Arguments are not properly created.
e.g., constant values are used while values are required to be randomly generated.	
TypestateError	A JCA API object is not in the right state to invoke a certain method
e.g., a method *reset()* of a MessageDigest object is invoked before passing any information into it via calls to methods *digest* or *update*.	
IncompleteOperationError	Tasks are not completed using JCA API objects.
e.g., a *MessageDigest* object is instantiated by invoking the *getInstance* method, but no further method invocation on this object is performed. The digest task will therefore not be achieved.	
ForbiddenMethodError	Unrecommended API methods are invoked.
e.g., *PBEKeySpec(char[] password)* is one of the constructor of JCA API *PBEKeySpec* for deriving cryptographic keys from a given password. Since a key generated without *salt* has been proven to be weak, this constructor should be used in specific scenarios.	
NeverTypeOfError	Certain types are forbidden when storing sensitive information.
e.g., password value for *PBEKeySpec* should never be store as type *String*, but *char[]*. Since object *String* is immutable in Java, password information in this type cannot be explicitly freed from memory. The garbage collector is in charging of deleting it, yet it is unpredictable from a user standpoint, opening opportunities for password leakage.	

Finally, we have implemented a crawler for collecting on Google Play some metadata (e.g., category, rate, etc.) associated to AndroZoo apks. These metadata are leveraged in criteria for comprehensively dividing the dataset into relevant subsets for further investigations.

III. Scope of the Study

In this section, we state the study problem along with the research questions that we intend to investigate, and describe the dataset of the study.

A. Problem Statement and Research Hypothesis

Given a crypto-API usage location, it is possible, with security expertise, to assess whether there is a misuse or not. Manual specification of usage rules however is tedious to collect over a large set of APIs. Previous studies have also shown that using a majority voting on usage patterns to conclude on the correctness of crypto-API usages will lead

to poor results given that wrong usages are widespread in Android apps. Our intuition however is that, as time goes by, developers of a given app learn to fix API misuses. Thus, it should be possible, by analysing code updates in an app lineage (i.e., the series of apk versions released for a given app), to infer crypto-API usage rules.

Our hypothesis is thus that: *"updates in API usages across an app lineage will tend to fix misuses"*. Consequently, if an extensively large set of app lineages can be collected in the wild, it would be possible to retrieve a substantially large and diverse set of crypto-API usage fixes. Then, by assessing recurring patterns, we could infer API usage rules.

This work is about empirically assessing the validity of our hypothesis for the case of the JCA APIs within Android apps.

B. Research Questions

The empirical study mainly aims at (re)investigating the following questions:

1) *To what extent do Android developers misuse crypto-APIs?* The literature claims, often based on few example apps, that developers regularly make mistakes in using crypto-APIs. We attempt to provide a thorough picture of the state of crypto-API usages across a representative dataset of real-world Android apps.
2) *Are crypto-API usage updates fixing misuses?* Investigating actual API usages with an oracle, based on security expert manual specifications, will eventually help to conclude on the validity of our research hypothesis. We ensure in this study, that the cases of specific app categories (e.g., high rating apps, financially sensitive apps, etc.) are also analysed in comparison with the general trends.
3) *What are the impacts caused by API usage updates?* This question investigates how crypto-API usages get updated, in an attempt to derive explanations on the statistical results obtained in the previous question. Concretely, we study the proportions of updates that either successfully fix misuses, or (re)introduce mistakes in correct usages, or that fail to fix misuses.

C. Dataset

We leverage the largest repository of real-world Android apps, AndroZoo [17], to collect data for our experiments. AndroZoo is a growing repository where automated crawlers continuously harvest Android apps from various app markets including the official Google Play store. At the time of writing, it was reported to contain over 8 million Android apks [23], the most diverse dataset available to the research community. Since AndroZoo continuously collects any apks that it has never seen before, it generally includes successive versions (i.e., apk releases) of the same app, which are relevant for reconstructing app lineages.

1) App Lineage Reconstruction: To identify app lineages from AndroZoo, we follow the approach proposed by Gao *et al.* [24] and illustrated by the four steps in Fig. 1: (1) extraction of application IDs, (2) app clustering by certificate, (3) app clustering by the market, and (4) app sorting by version code. This process was applied on a snapshot of AndroZoo in September 2018. Out of the 8 million apps, we only considered lineages which include at least 10 apk releases. The lineage reconstruction yielded 43 365 app lineages accounting for 745,101 apks. This lineage dataset is twice as large as the dataset presented by Gao *et al.* in [24] .

2) Misuse detection: We assess crypto-API misuses based on the reports of the $CogniCrypt_{SAST}$ static checker. This tool, which implements analysis based on expert manual specifications of API usage rules, is used to collect the oracle to support our empirical assessment. Analyses are performed on a High-Performance Computing (HPC) platform [25]. Overall, we leveraged 142 HPC instances, each utilizing 24GB of memory, to successfully parse all 745 thousands apks in 5 days.

Fig. 1: App Lineages Re-construction Process [24].

3) Dataset Curation: We took steps to remove from our study all irrelevant cases of apks or misuses reported by $CogniCrypt_{SAST}$.

- Apk releases with no JCA API usages are excluded from our study. Thus, 18% of our initial dataset is left out.
- Apk releases on which $CogniCrypt_{SAST}$ fails to generate a final report are also dropped from the study. Such cases often occur when the process runs out of memory. While the recommended memory size for $CogniCrypt_{SAST}$ is 8GB, we allocate 24GB in our experiments. Nevertheless, a few apk analyses are not able to be completed.
- Obfuscated apk releases are left out from the study. Since developers recurrently rely on obfuscation techniques to prevent reverse engineering of their apps, static checkers such as $CogniCrypt_{SAST}$ are challenged in their analyses: $CogniCrypt_{SAST}$ reports '?' for erratic character series that appear as class names. Given that our study of API misuses leverages class names as the basic unit for localisation, such unidentified class names constitute noise. Thus, after analysis, when the generated report contains any unlocalised class name, the corresponding apk is dropped.

TABLE III: Number of APKs and lineages in the Dataset

	# lineages	**# APKs**
initial dataset	43 365	745 101
remove because JCA API is not used	-3 882	-135 752
remove due to CogniCrypt failures	-108	-9 374
remove because of obfuscation	-7	-1 100
remove due to combination of the 3 conditions[1]	-155	
final dataset	39 213	598 875

Eventually, 146,226 apks are excluded from the dataset of apks. Table III summarizes some statistical details about our dataset. An apk is removed when the one of the situations above occurs. However, an app lineage is only removed when all its app versions are excluded.

Fig. 2 shows the distribution of the dex size of apks for the initial and the final dataset. Our statistical tests indicate

[1]e.g., an app lineage of 10 app versions, 5 app versions could be removed because of tool failure while the rest could be caused by obfuscation

no difference between the two distributions, implying that our final dataset is still representative of the initial dataset (at least w.r.t app sizes).

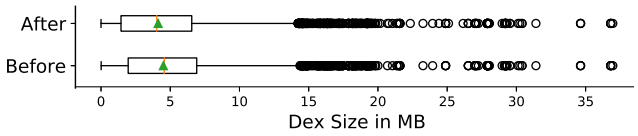


Fig. 2: Distribution of Dex Size before-after dataset curation

Metadata Collection. Some of our investigations require up-to-data metadata (e.g., category, rating, number of installs) from markets. Because *GooglePlay* implements location restrictions (only apps targeting a country's users are made visible in that country), we were able to collect metadata for around 60% of the lineages.

IV. Methodology of the Study

We carry experiments to assess the hypothesis behind relying on usage updates to mine crypto-API usage rules. We explore usage updates:

- with an analysis of *pairwise* comparisons among apks successive releases within the lineages in our dataset;
- with an overall *lineage-wise* study of the recurrence of misuses in an app across its entire lineage.

We further investigated updates for *selected subsets* of apps to confront the general trends against specific cases for popular apps, or sensitive apps.

A. Pairwise comparisons of apks from the same lineage

Given an app lineage l_i, its associated apk releases are combined into pairs (apk_{j-1}, apk_j), where $1 < j \leq n$, for the purpose of checking differences between misuses in apk_j and apk_{j-1}. The comparison takes into account only usage instances that are found at the same *code location* in both apps and that are relevant to the same *API*. In this study, a *code location* (lo) is represented by both the class and method in which the API usage is found. We consider the following cases which may occur:

- *misuse fixing (MF)* update: the $CogniCrypt_{SAST}$ analysis flags an issue with a usage in apk_{j-1} but not with the usage at the same location in apk_j. We conclude that the misuse has been fixed by the update.
- *misuse introducing (MI)* update: the $CogniCrypt_{SAST}$ analysis flags an issue with a usage in apk_j but not with the usage at the same location in apk_{j-1}. In contrast to a *MF* update, such an update introduces misuses.
- *misuse fixing and introducing (MFI)* update: the $CogniCrypt_{SAST}$ analysis flags an issue with a usage at a given location in both apk_j and apk_{j-1}, but the misuses are different. This suggests that developers corrected the previous misuse during the update, but somehow made another mistake.
- *none update*: the $CogniCrypt_{SAST}$ analysis flags the same issue with a usage in apk_{j-1} and apk_j at the same location. We conclude that the developers did not notice the issure during app updates.

To distinguish among different usages of the same API at the same location, one should take into account variables names associated to instantiated objects from API classes. In practice, however, the reverse-engineering of apks assigns random names to variables, making these names differ across the pair of apps. We use simple heuristics to match relevant pair of usage instances and iteratively start with identifying *none update* cases, then *MFI* updates, before *MF* updates and *MI* updates.

MI update and *MFI update* constitute two cases of *mis-updates*, as they result in API misuses in the most recent version of the app.

B. Investigations of updates across lineages

We investigate the overall evolution of a given app w.r.t. its misuses of crypto-APIs. We then study the trends of usage issues in app lineages. To that end, first, for each app version apk_j in an app lineage l_i, we compute a *misuse ratio* r_j defined in equation 1.

$$\begin{aligned} m_j &\leftarrow total\_number\_API\_misuses(apk_j) \\ u_j &\leftarrow total\_number\_API\_usages(apk_j) \\ r_j &:= \frac{m_j}{u_j} \end{aligned} \tag{1}$$

We consider the ordered ratio list R_i of app lineage l_i as $R_i := \{r_1, r_2, \cdots, r_n\}$, a list of misuse ratio of all apk releases included in l_i with r_1 being the misuse ratio of the first apk of l_i, r_2 the misuse ratio of the second apk, etc. We then compute the slope $s_i = linear\_regress(R_i)$ of the regressed between all points of R_i. In this study, s_i is used to characterize the *misuse trend*: a negative s_i indicates that lineage l_i is evolving towards a better usage of crypto-APIs, while a positive s_i suggests that the usage of crypto-APIs is worsening with app updates. A null s_i indicates a status quo. The bigger the value of $|s_i|$, the faster the evolution in a lineage.

V. Study Results

We now provide experimental results obtained while investigating the research questions enumerated in Section III-B. In particular, we show statistics on crypto-API misuses in the wild, summarize the success rates in API misuse updates between apk releases, reveal the evolution of misuses across lineages, and eventually discuss the difference between the whole dataset and selected categories of apps.

A. RQ1: Crypto-API misuses in Android apps

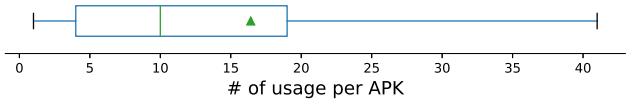


Fig. 3: Distribution of JCA API usages in the study dataset

Crypto-APIs are widely used in Android apps. Statistics presented earlier (Table III) indicated that over 80% (or 598 thousands apks) of the app versions in our sample dataset of 745 thousands apks include code with JCA API usages. Fig. 3 further shows the distribution of JCA API usages in our dataset of 598k apks. The usage statistics are collected from the analysis reports of $CogniCrypt_{SAST}$. On median

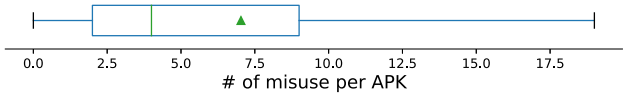
average, 10 JCA API usage instances can be identified per app, while 75% of apks include at least 5 usage instances.

Table IV summarizes the statistics on API misuses among the apks that include JCA API usages. 96% of apks include misuses, and 97.6% of lineages include at least an apk version with a misuse. On average 1 misuse is found among 6 usages of JCA APIs.

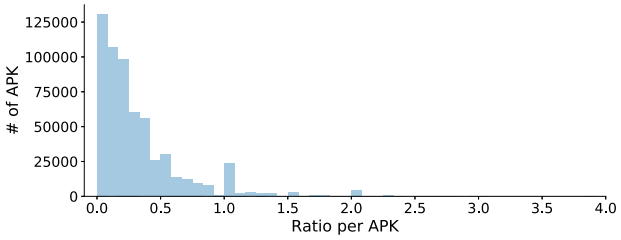
TABLE IV: Statistics on API Misuses in the dataset apks

Types	Numbers	Related Information
Percentage of APKs with misuse	96%	number of APKs without misuses is 24,880
Percentage of app lineages with misuse	97.6%	number of lineages without misuses is 942
misuse to usage ratio	1 : 5.53	total number of API usages is 23,281,216 and misuses is 4,210,667

We detail the misuse spread in Fig. 4. As shown by the misuse distributions in apks represented in Fig. 4a, a given apk contains commonly between 2 to 9 misuse cases. On median average, 4 misuses can be found per app. We further show in Fig. 4b the distribution of the ratio (as defined in Eq.1) between misuses and usages within apks. Interestingly, there are cases with the ratio is bigger than 1, i.e., there are more misuse instances than usages. This suggests that developers can make several mistakes when using a single crypto-API.



(a) Distribution of # of Misuse per APK



(b) Distribution of Misuse to Usage Ratio per APK

Fig. 4: API Misuse Distribution

The preponderance of the different APIs to be affected by misuses is detailed in Fig. 5. Over 60% of apks include instances of the *MessageDigest* crypto-API with a misuse. This is due to the widespread issue of using weak hashing algorithms. While the percentage for *Cipher*, the API in the second place, is around 21% for which the main misuse is incompleted operation indicating that some further method calls (e.g., *update*, *doFinal*) are expected but never achieved.

Table V further provides statistical details on the types of errors that are raised by $CogniCrypt_{SAST}$ as well as on the top API methods that are concerned by misuses. Method *getInstance(java.lang.String)* of crypto-API *MessageDigest* takes as argument a String specifying the hashing algorithm. This algorithm must offer a strong protection and thus should be one of the recommended algorithms (e.g., *SHA-256*, *SHA-384* and *SHA-512*). When weak algorithms are used (e.g., *MD5* or *SHA-1*), it represents a *ConstraintError* which makes the app exposed to attacks.

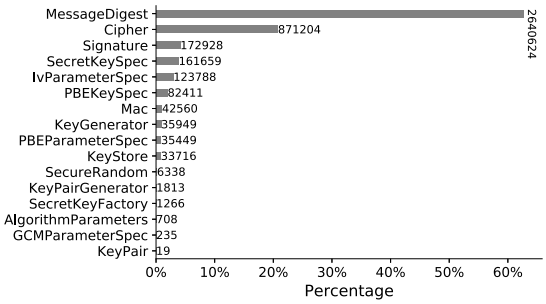


Fig. 5: Misuse Ranking based on JCA APIs

TABLE V: Ranking of Misuses

Ranking by Types		
Type	**%**	**Misuse #**
ConstraintError	60.93%	2,565,892
RequiredPredicateError	14.63%	615,921
TypestateError	12.27%	516,758
IncompleteOperationError	11.64%	490,157
ForbiddenMethodError	0.51%	21,565
NeverTypeOfError	<0.01%	374
Top 10 by API Methods		
API Method	**%**	**Misuse #**
MessageDigest.getInstance(java.lang.String)	48.43%	2,039,428
Cipher.getInstance(java.lang.String)	8.80%	370,354
MessageDigest.reset()	8.06%	339,305
SecretKeySpec.<init>(byte[],java.lang.String)	3.79%	159,594
MessageDigest.digest()	3.78%	159,205
Cipher.init(int,java.security.Key, ...) [2]	2.95%	124,274
IvParameterSpec.<init>(byte[]) [3]	2.92%	122,749
Cipher.init(int,java.security.Key)	2.50%	105,208
Signature.getInstance(java.lang.String)	1.99%	83,704
Signature.initVerify(java.security.PublicKey)	1.80%	75,705

Nevertheless, we find that although *MessageDigest* is the API with the overall highest number misuses, it is not the most misuse-prone API. Crypto-API *PBEKeySpec* has a higher misuse ratio. Table VI ranks the JCA APIs based on such a ratio. We provide details of misuse ratio distributions for each API in Fig. 6. The median value of misuse ratio for most of APIs is 0. For these APIs, only *IvParameterSpec*, *KeyGenerator* and *SecretKeySpec* are showing clear boxes and whisker lines while the rests only show outliers (which is not shown in the figure for clearity reason). This suggests that mistakes are quite rare for such APIs. In contrast, 5 APIs show relatively high misuse ratios. Based on their median values, *PBEParameterSpec* is the most error-prone (0.67), followed by *PBEKeySpec* (0.5), *MessageDigest* (0.24) and *Cipher* (0.9).

We further investigated *PBEKeySpec* and *PBEParameterSpec* as they stand out in terms of misuse ratio distributions. For *PBEKeySpec*, we found that the main misuse type is *IncompleteOperationError* which is mainly caused by missing of calling method *clearPassword()* to clear the password from the memory. *PBEParameterSpec* misuses only occur with type *ConstraintError*(80%) or *RequiredPredicateError* (20%). *ConstraintError* generally refers to the small iteration numbers as discussed in Listing 1.

[2]init(int,java.security.Key,java.security.spec.AlgorithmParameterSpec)

[3]Note: method *<init>* is the constructor of the class while method *init* is a common method of the class

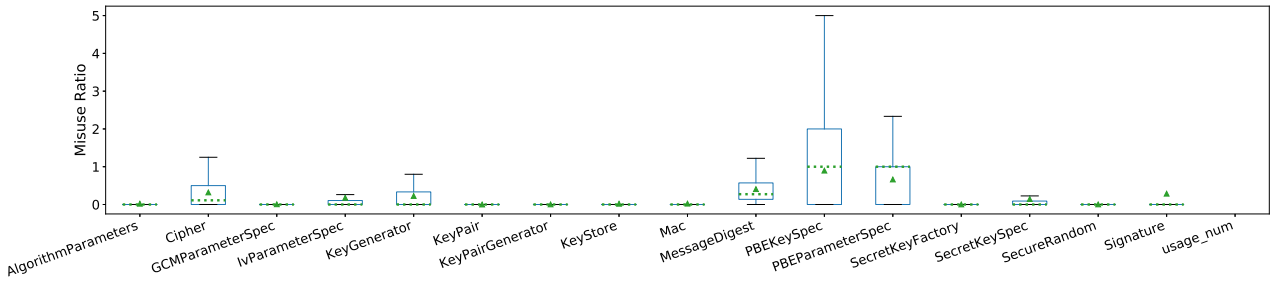


Fig. 6: Misuse Ratio Distribution based on APIs

Three JCA APIs (namely, *SecretKey*, *HMACParameterSpec* and *DSAGenParameterSpec*) are missing from the API usage list of our dataset. Four other APIs (namely *DHGenParameterSpec*, *DHParameterSpec*, *DSAParameterSpec*, and *RSAKeyGenParameterSpec*) although they have usage cases in our dataset, no misuses have been reported for them. Further investigations suggest that these four APIs are actually seldom used, and their usage rules are in any case straightforward. Indeed, except *DHParameterSpce*, these APIs are ranked at the bottom of the API usage ranked list. Their usage rules are only about the invocation of constructors with a number of constraints. Although other APIs, such as *GCMParameterSpec* and *KeyPair*, having similar simple rules, can be found with misuse cases, the occurrences are very low. In conclusion, these findings suggest that, since developers cannot avoid using crypto-APIs, simplifying the usage rules during API design could be an effective way to avoid misuses.

TABLE VI: JCA API Misuse to Usage Ratio Ranking. The misused apk % is calculated by number of apks containing the misuse divided by number of apks containing the relevant API usage.

API	Misuse Ratio	Misused APK %
PBEKeySpec	0.462098	55.50%
MessageDigest	0.287114	93.02%
Cipher	0.271808	50.77%
PBEParameterSpec	0.251944	57.26%
Signature	0.204835	23.76%
KeyGenerator	0.171380	27.74%
IvParameterSpec	0.102394	27.58%
SecretKeySpec	0.070372	27.81%
Mac	0.067249	3.05%
KeyStore	0.028322	9.12%
GCMParameterSpec	0.008141	0.79%
KeyPairGenerator	0.005865	0.60%
SecretKeyFactory	0.003974	0.35%
AlgorithmParameters	0.003307	3.46%
SecureRandom	0.003244	0.94%
KeyPair	0.000019	0.01%

The JCA APIs for implementing cryptography are widely misused across Android apps. Usage mistakes range from issues with parameter initialisation to mishaps with the sequence of API method invocations. Nevertheless, all crypto-APIs are not similarly affected by misuse cases.

B. RQ2: Impact of crypto-API usage updates on misuse cases

From our dataset of 39,213 lineages accounting for about 598K apks, we collected 559,662 apk pairs (apk_{j-1}, apk_j), where apk_j is the updated version of apk_{j-1}. From these apk pairs, we were able to extract 3,291,723 crypto-API usage pairs that fall into the four categories defined in Subsection IV-A: *MF* update, *MI* update, *MFI* update, and *none* update. Among 559,662 apk pairs, around 75% (or 410,587) of them fall into the *none* update category. This situation is even worse if we count the *none* update rate at the crypto-API uasge pair level, over 95% of the 3,291,723 misuses are not touched by app developers during the evolution of Android apps. This finding suggests that app developers are unlikely to update crypto-API usages when updating their apps or may not be even aware of the misuses in their app code.

For the remaining 162 970 crypto-API usage pairs involved with developer updates (e.g., not in the *none* update category), surprisingly, only 76,341 of them (less than 47%) have successfully fixed the misuse issues (i.e., falling into the *MF* update category). Over half of the crypto-API updating attempts fall into either *MI* update category (e.g., 72,143, around 44%) or *MFI* update category (e.g., 14,486, around 9%). At the apk level, the *MF*, *MI*, and *MFI* attempts are 53,030, 50,183, and 4,483, respectively, resulting in still over 50% of mis-update rate. This surprising result indicates that even in the cases that app developers are aware of crypto-API misuse and are attempting to fix such misuses, most of them do not have the right knowledge to properly fix the misuse issues. As a result, our previous assumption on mining crypto-API usage rules from the evolution of Android apps cannot be easily realised in practice.

Because the overall dataset leveraged, although very big, is collected from various sources, we hypothesise that the selected datasets (or app lineages) contain a broad set of apps with varying quality. Consequently, the large mis-update rate might be impacted directly by the selection of poor quality apps. If we focus our experiments on high-quality apps only, we might be able to observe clear trends that app developers are recurrently and successfully fix crypto-API misuse issues. To this end, we resort to two specific subsets to reconduct our empirical experiments.

- **Reputed Apps.** Updates of reputed apps are selected only within lineages where the app has high rates (i.e., ≥ 4.5) and large installs (i.e., $\geq 1\,000\,000$). With this subset, we assess whether widely used apps are similarly affected by crypto-APIs misuses.
- **Finance Apps.** Updates of finance apps are focused on app lineages tagged in *GooglePlay* as being for financial services. In this case, we again constrain this selection to apps with high rates and large installs as the case of Reputed apps. Because finance apps are critical to security

issues, with this subset, we assess whether app developers of finance apps have special treatment to crypto-API misuse.

Table VII summarises the experimental results we observed for the different sub-datasets. For comparison purpose, we also present the *Overall* results representing the statistics computed for the whole dataset of apks. Following the same strategy, we also provide data on the number of *MF*, *MI* and *MFI* updates. The statistics are computed at the apk level (i.e., given a pair of successive apks in a lineage, how many of these pairs contain at least one *MFI* update, at least one *MF* update, etc.) and at the usage level (how many updates turned out to be a *MF*, a *MFI* case, etc., i.e., we count the number of *MF* update, *MFI* update, etc.). Unfortunately, compared to the mis-update rates of the *Overall* set, the results observed on that of the selected subsets do not suggest any substantial difference, only about 1 out of 2 updates will yield a correct fix for a crypto-API misuse.

TABLE VII: Misuse Update Statistics

		MF	MI	MFI	Mis-update Rate
Overall	APK level	53,030	50,183	4,483	50.76%
	Usage level	76,341	72,143	14,486	53.16%
Reputed	APK level	4,300	3,934	446	50.46%
	Usage level	5,809	5,204	1,862	54.88%
Finance	APK level	179	153	30	50.55%
	Usage level	232	192	195	62.52%

Fig. 7 further illustrates the distribution of slopes (i.e., misuse trends as defined in Subsection IV-B) across the different subsets. We consider the evolution of misuse rate across each app lineage. When the mis-update rate is stable across the lineage, the slope metric evaluates to 0. A negative slope implies that the situation is getting better along the lineage: latest updates in the lineage are more successful. In contrast, a positive slope suggests that the situation is getting worse along the lineage.

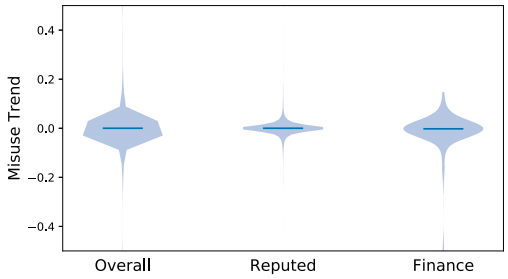


Fig. 7: Distribution of Slopes (Misuse Trend)

We observe that, for all three datasets, the distribution of trend slopes presents more or less a symmetric pattern around a median value at 0. The fact that the distribution spans are relatively narrow further suggests that for most lineages, crypto-API usage updates are rather stable. In the overall dataset, we can find 10,316 (26.31%) lineages with a negative trend of misuse update. Nevertheless, the successful cases in recent updates are still not more numerous: the mis-update rate amounts to 53.66% when we consider these lineages apps altogether.

Based on Table VII, it is noteworthy that the *MI* update (i.e., where a mistake is made on a usage that was previously correct) cases are the major contributors to the *mis-update* rate. There are often as many *MI* updates as *MF* updates in each dataset. We further investigate the recurrence of misuses across the different datasets and found 19,392, 1,332 and 65 recurrent misuse cases respectively in the overall, reputed and finance datasets. Concretely, we consider a misuse to be recurrent when, within an app lineage, an update has eliminated it (i.e., *MF* update), and then it has been later introduced (i.e., *MI* update). Overall, we find that 25% of *MI* updates actually represent recurring misuses, suggesting that the associated *MF* updates may have been unintentional (which made them likely to be reintroduced).

Table VIII details the misuse update results for the different APIs with their rankings. Even at the level of each API, *MF* and *MI* updates appear to compensate each other. This can be observed by the similar numbers of updates as well as their ranks across the API list, with APIs *Cipher* and *MessageDigest* leading the statistics (consistently with the misuses preponderance shown in Table V).

TABLE VIII: Misuse Update Ranking by APIs

API	MF		MI		MFI	
	#	Rank	#	Rank	#	Rank
Cipher	20,574	1	21,751	1	2,549	2
MessageDigest	17,668	2	20,451	2	1,283	3
IvParameterSpec	14,300	3	14,150	3	17	10
SecretKeySpec	14,085	4	14,126	4	71	7
KeyGenerator	3,372	5	3,465	5	65	8
Mac	878	6	860	6	9,382	1
Signature	504	7	516	7	864	4
KeyStore	278	8	268	9	153	5
PBEKeySpec	155	9	304	8	83	6
SecureRandom	131	10	180	11	0	12
PBEParameterSpec	123	11	212	10	0	12
AlgorithmParameters	47	12	22	12	2	11
KeyPairGenerator	14	13	20	13	0	12
GCMParameterSpec	13	14	13	14	0	12
SecretKeyFactory	1	15	3	15	17	10
KeyPair	0	16	0	16	0	12

Statistical data in Table VIII show that none of these three update kinds happened with *KeyPair* as shown in the last line of the table. This is likely due to its low misuse occurrences (i.e., 19 according to Table V). Moreover, API *SecureRandom*, *PBEParameterSpec*, *KeyPairGenerator* and *GCMParameterSpec* do not show any cases of *MFI* updates. Nevertheless, we still cannot learn from their changes as they can either be *MF* or more like *MI* updates.

> *Android app developers are generally unaware of crypto-API misuses and hence will unlikely fix such issues. Unexpectedly, usage updates are evenly distributed between successful fixes and failures. The recurrence of misuses further imply that most of the successful updates may have not been made intentionally.*

C. RQ3: Errors and methods impacted by misuse updates

We now investigate the types of errors that are concerned by *MF*, *MI* and *MFI* updates. Table IX indicates that *MF* updates are dominated by misuses cases of type *RequiredPredicateError* (67%) followed by *ConstraintError* (29%). Consistently with the previous finding on recurrence of misuses, we note that *MI* updates follow a similar pattern. We recall that

both *RequiredPredicateError* and *ConstraintError* generally concern the initialization or the selection of proper arguments to API methods.

TABLE IX: Misuse Update Ranking by Types

API	MF		MI		MFI	
	%	Rank	%	Rank	%	Rank
RequiredPredicateError	66.78%	1	69.99%	1	3.86%	4
ConstraintError	29.10%	2	25.73%	2	7.08%	3
TypestateError	3.35%	3	3.38%	3	12.07%	2
IncompleteOperationError	0.71%	4	0.84%	4	76.67%	1
NeverTypeOfError	0.05%	5	0.05%	5	0.05%	6
ForbiddenMethodError	0.02%	6	0.01%	6	0.27%	5

MFI updates show a different pattern, where *IncompleteOperationError* becomes the major misuse type: in 77% of cases, the update does not properly fix the errors. *TypeState* misuses then account for 12% of misuses that are difficult to fix. Both are about the sequence of API method invocations, either about missing certain method calls or related to a wrong order of invocation. For example, we note that in several cases, developers invoke the method *doFinal* immediately after getting an object instance of the *Mac* API class. However, they are supposed to start with the invocation of method *init* and, occasionally perform an *update* before calling *doFinal*: this leads to a misuse of type *TypestateError*. During the updates, it appears that most developers are trying to fix the problem by adding invocations of method *update*. Nevertheless, they generally still do not call *init* which leaves the issue improperly addressed. More strangely, we noted that in many cases, instead of further completing the fix, developers simply reverted back to the previous misuse version. *IncompleteOperationError* is seen an opposite scenario: after generating the instance of *Mac*, method *doFinal* is never called. Developers update the usage by adding method calls such as *init* or *update* but never add *doFinal* call, which again leaves this issue unresolved.

TABLE X: The Most Common MFI Update Methods

API	Missing Method		#	%	Explanation
	From	To			
Mac	*update* or *doFinal*	*init*	4,182	28.87%	before update, *init* was call but missing method call of *update* or *doFinal*. However, update even removed method call of *init*.
Mac	*init*	*update* or *doFinal*	3,971	27.41%	expected method call of *init* was added during update but still require further method call of *update* or *doFinal* which is missing.

From the perspective of crypto-API methods, Table XI exhibits details about the top API method invocations which are involved in *MF* or *MI* updates. Note that the top 9 methods for the two kinds of updates are exactly the same. The last two lines of the table are the 10th methods for the two update kinds. As the most commonly misused method (cf., Table V), *getInstance* of *MessageDigest* is also the most often updated method for fixing but also for introducing misuses. The recurrent misuse with this method is about the parameter specifying the hashing algorithm. Weak algorithms such as *SHA1* and *MD5* or sometimes even wrong values, like *SHA*, are used.

Finally, *MFI* updates are generally related to *IncompleteOperationError* and *TypestateError* types, which are all caused by incorrect API method invocation sequences. Due to space limitation, Table X only shows a couple of example cases of how *MFI* updates occur. These examples show that misuses can be bounced back and forth when API usages require several steps in the invocation sequence.

TABLE XI: Top 10 MF & MI Update Methods

API Method	MF			MI		
	#	%	Rank	#	%	Rank
MessageDigest.getInstance(java.lang.String)	19,183	25.13%	1	16,331	22.64%	1
Reason: parameter with value: MD5, SHA1, SHA						
IvParameterSpec.<init>(byte[])	14,070	18.43%	2	14,234	19.73%	2
Reason: parameter is not randomized						
SecretKeySpec.<init>(byte[],java.lang.String)	13,681	17.92%	3	13,636	18.90%	3
Reason: first parameter is not randomized						
Cipher.init(int,java.security.Key, java.security.spec.AlgorithmParameterSpec)	12,207	15.99%	4	12,097	16.77%	4
Reason: second parameter is not properly generated						
Cipher.init(int,java.security.Key)	6,559	8.59%	6	6,274	8.70%	5
Reason: second parameter is not properly generated						
KeyGenerator.init(int,java.security.SecureRandom)	3,363	4.41%	6	3,285	4.55%	6
Reason: second parameter is not properly randomized (e.g., fixed seed)						
Cipher.getInstance(java.lang.String)	2,553	3.34%	7	1,888	2.62%	7
Reason: parameter with not recommended algorithm (e.g., DES), unappropriated combination of algorithm and feedback mode (e.g., AES/EBC) or with without padding scheme						
MessageDigest.reset()	604	0.79%	8	630	0.87%	8
Reason: missing method call of *digest* or *update*						
SecretKeySpec.<init>(byte[],int,int, java.lang.String)	445	0.58%	9	449	0.62%	9
Reason: first parameter is not randomized						
Mac.doFinal()	366	0.48%	10			
Reason: expected to call method *init* before.						
Signature.initSign(java.security.PrivateKey)				391	0.54%	10
Reason: private key (first parameter) is not properly generated						

Misuses caused by missing steps when using crypto-APIs appear to be difficult to fix. In turn this difficulty is manifested by recurrent failures in API usage updates.

D. Discussion

Initially, we planned to build on the assumption that app developers are likely to fix crypto-API misuse issues during app evolution. Hence, by mining lineages of a large set of Android apps, one can summarise the crypto-API usage rules. Unfortunately, and also surprisingly, our investigations reveal that :

- crypto-API misues are very common in Android apps.
- app developers are not likely to fix misuses when they update app code.
- For the cases where developers try to fix such misuses, they are often not able to make correct fixes.
- some misuses are recurrently fixed and reintroduced, implying that most of the successful updates might not be performed intentionally to fix the relevant misuses.
- some APIs are more impacted by misuses than others. Misuse updates are however likely to fail as much as to succeed.

We showed that even reputable or sensitive apps are substantially suffering from crypto-API misuses, suggesting that our community is still lacking reliable means to address this problem. Therefore, immediate actions are needed. From app developer side, the recurrence of misuses suggests a need to provide better developer education on how to correctly use crypto-APIs. Similarly, crypto-API providers also need to find better ways to design crypto-APIs to reduce the error margins. Finally, app markets must pay special attention to such apps

with misused crypto-APIs, which will create a momentum of developers addressing them seriously.

More concretely, developers should pay extra attention when using *MessageDigest*, *PBEKeySpec* and *Mac*, as they are the widest misused, most misuse-prone and most difficult to be corrected APIs respectively. Choosing algorithms like *SHA-256* instead of *SHA-1* or *MD5* can quickly avoid most of the misuses in *MessageDigest*. While generating salt randomly and remembering to call method *clearPassword()* at the end can make usage of *PBEKeySpec* safe and sound. Finally, the key to use *Mac* correctly is the order of method invocations. Methods *getInstances* and *init* should be always used in the first 2 steps. Method *update* could be invoked more than once afterwards. And *doFinal* should always be called once at last.

VI. Threats to Validity

For internal threats to validity, our results may be impacted by the dataset selected, which might not be representative. We attempt to mitigate this impact by considering a large set of Android apps. Furthermore, the app versions in a lineage are sequenced based on their version code, which however may not be always true as the version code is configured by app developers. We did not however find any false positives by sampling lineages.

Regarding external threats to validity, our results may be impacted by the false alarms of $CogniCrypt_{SAST}$. We have actually benchmarked a set of apps to check the false positive rates of $CogniCrypt_{SAST}$. Our manual verification confirms that $CogniCrypt_{SAST}$ is effective. We have only observed one case where $CogniCrypt_{SAST}$ may yield false positives, which is related to the artificially created *dummyMainMethod* method (because Android apps do not have a single *main()* like traditional Java code). We have reported this issue to the authors of $CogniCrypt_{SAST}$ and excluded such cases from consideration in this work.

Furthermore, negative result normally refers to a statistical null hypothesis is accepted or an approach is not better than the baseline. While, in this work, we use this term to emphasise the difference between the assumption and the surprising empirical results. Meanwhile, although we investigated our assumption from serval different angles, we did not exhaust all possible ways. Therefore, there are still chances for the assumption to be ture for certain elaborate sub-datasets.

Finally, we have only conducted our experiments on APIs and a few sub-datasets of apps. It might be still possible to mine usage rules on other datasets or other crypto-APIs. More sophisticated approaches may be successful for mining crypto-API usage patterns from the evolution of Android apps.

VII. Related Work

Crypto-APIs have become a major feature in modern programming languages for encrypting/decrypting sensitive messages, while the misuses of such APIs have also been extensively studied in our community. In this section, we briefly discuss the representative ones.

Misuses of crypto-APIs. CryptoLint is a tool that performs lightweight syntactic analyses for pinpointing violations of hard-coded crypto-API usage rules in Android apps [26]. Similar to CryptoLint, Crypto Misuse Analyzer (CMA) [27] is also based on hard-coded rules to flag misuses of crypto-APIs. In this work, we leverage $CogniCrypt_{SAST}$ to detect misuses of crypto-APIs in Android apps. To the best of our knowledge, $CogniCrypt_{SAST}$ is so far the most advanced tool for detecting misuses of crypto-APIs. Indeed, the rules hard-coded in CryptoLint and CMA are also contained in the rules of $CogniCrypt_{SAST}$. Therefore, the misuses of crypto-APIs leveraged in this work should be representative and suitable for this study. Also, by manually exploring 49 Android apps, Chatzikonstantinou et al. [28] confirm that at least 88% of the studied apps have misused at least one crypto-API. The ratio obtained in this work is even slightly higher, showing that misuses of crypto-APIs are indeed very common in Android apps.

Mining Usage Patterns in Android Apps. Researchers have reported various pattern mining approaches in the field of Android analyses. For example, Linares-Vasquez et al. [29] have conducted an empirical investigation to mine the energy-greedy API usage patterns in Android apps as well as mine the app usages for generating actionable GUI-based execution scenarios [30]. Similarly, Karim et al. [31] mine Android apps for recommending permissions while Moonsamy et al. [32] aim at mining permission patterns for contrasting clean and malicious Android apps.

Android App Evolution Analysis. In this work, we leverage app lineages to understand the evolution of Android apps w.r.t. the usage of crypto-APIs. Android app evolution analysis is not new. Researchers have presented various studies for understanding the evolution of Android apps [33], [34], [35]. For example, Gao et al. [24] have presented an empirical study aiming at understanding the evolution of Android app vulnerabilities. Similarly, Taylor and Martinovic [36] also investigate the evolution of security and privacy issues in Android apps. These two approaches, although conducted separately, have both shown that Android apps do not become safer over several years of evolution, which is in line with the major finding in this work, i.e., the misuses of crypto-APIs are not likely to be fixed by app developers.

VIII. Conclusion

Crypto-API misuses are common in Android apps. Mining usage rules is thus challenging given the noise in developer code. We hypothesise in this paper that usage updates are likely fixing misuses, and may thus be efficiently leveraged for mining usage rules. We perform a large-scale investigation of thousands of Android app lineages and fail to confirm our initial hypothesis. We report these negative results to the community and make available the artefacts of the study.

Availability: https://negative-crypto-api-mining.github.io/

Acknowledgment

This work is supported by the Luxembourg National Research Fund (FNR) through grant PRIDE15/10621687/SPsquared and project CHARACTERIZE C17/IS/1169386.

REFERENCES

[1] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

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

[3] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, pages 83–90, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[4] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew 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*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[5] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[6] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. In *10: 1-10: 27*, 2018.

[7] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto api rules from code changes. *SIGPLAN Not.*, 53(4):450–464, June 2018.

[8] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 711–722, New York, NY, USA, 2016. ACM.

[9] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 57–72. ACM, 2001.

[10] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[11] Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 43–52. IEEE, 2009.

[12] Tegawendé F Bissyandé, Laurent Réveillère, Julia L Lawall, and Gilles Muller. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 60–69. ACM, 2012.

[13] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2017.

[14] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946, New York, NY, USA, 2016. ACM.

[15] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 709–712, Piscataway, NJ, USA, 2015. IEEE Press.

[16] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[17] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *MSR '16 Proc. of the 13th Intl. Conference on Mining Software Repositories*, pages 468–471, Austin, Texas, May 2016.

[18] Oracle. Java cryptography architecture (jca). https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html.

[19] Apache. Apache commons crypto. https://commons.apache.org/proper/commons-crypto/index.html.

[20] CogniCrypt Developers. $cognicrypt_{SAST}$. https://github.com/CROSSINGTUD/CryptoAnalysis.

[21] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 931–936, Piscataway, NJ, USA, 2017. IEEE Press.

[22] CogniCrypt Developers. Cognicrypt for android. https://github.com/CROSSINGTUD/CryptoAnalysis-Android.

[23] AndroZoo. Androzoo web site. https://androzoo.uni.lu/.

[24] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. On vulnerability evolution in android apps. In *The 40th International Conference on Software Engineering, Poster Track (ICSE 2018)*, 2018.

[25] Anonymised for blind review. Anonymised for blind review. In *Anonymised for blind review*, pages xx–yy. xxx, xxxx.

[26] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher 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.

[27] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi 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.

[28] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*, pages 83–90. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[29] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM, 2014.

[30] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 111–122. IEEE, 2015.

[31] Md Yasser Karim, Huzefa Kagdi, and Massimiliano Di Penta. Mining android apps to recommend permissions. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 427–437. IEEE, 2016.

[32] Veelasha Moonsamy, Jia Rong, and Shaowu Liu. Mining permission patterns for contrasting clean and malicious android applications. *Future Generation Computer Systems*, 36:122–132, 2014.

[33] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[34] Paolo Calciati and Alessandra Gorla. How do apps evolve in their permission requests?: a preliminary study. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 37–41. IEEE Press, 2017.

[35] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[36] Vincent F Taylor and Ivan Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 45–57. ACM, 2017.