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 usage updates 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 misuse 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-API 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 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 [13], 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.

<table>
<thead>
<tr>
<th>API Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.security.AlgorithmParameters</td>
<td>maintainer for security parameters for specific algorithms</td>
</tr>
<tr>
<td>javax.crypto.Cipher</td>
<td>provide encryption and decryption functionality</td>
</tr>
<tr>
<td>javax.crypto.spec.DHGenParameterSpec</td>
<td>parameters for generating Diffie-Hellman parameters for DH key agreement</td>
</tr>
<tr>
<td>javax.crypto.spec.DHParameterSpec</td>
<td>parameters used for DH algorithm</td>
</tr>
<tr>
<td>javax.security.spec.DSAParameterSpec</td>
<td>parameters used for DSA parameter generation</td>
</tr>
<tr>
<td>javax.crypto.spec.GCMParameterSpec</td>
<td>parameters for cipher using Galois/Counter Mode</td>
</tr>
<tr>
<td>javax.xml.crypto.dsig.spec.HMACParameterSpec</td>
<td>parameters for the XML signature HMAC algorithm</td>
</tr>
<tr>
<td>javax.crypto.spec.IVParameterSpec</td>
<td>Initialization Vector for block cipher</td>
</tr>
<tr>
<td>java.security.KeyPair</td>
<td>generate keys for encryption-decryption</td>
</tr>
<tr>
<td>java.security.KeyPairGenerator</td>
<td>create public/private key pairs</td>
</tr>
<tr>
<td>java.security.KeyStore</td>
<td>a memory storage to maintain keys and certificates for later usage</td>
</tr>
<tr>
<td>java.security.MessageDigest</td>
<td>Message Authentication Code for message integrity protection</td>
</tr>
<tr>
<td>javax.crypto.spec.PBEKeySpec</td>
<td>specification of a Password Based Encryption key</td>
</tr>
<tr>
<td>javax.crypto.spec.PBEParameterSpec</td>
<td>parameters for password based encryption</td>
</tr>
<tr>
<td>javax.crypto.spec.RSAKeyGenParameterSpec</td>
<td>parameters for RSA key pair generation</td>
</tr>
<tr>
<td>javax.crypto.SecretKey</td>
<td>a symmetric secret key</td>
</tr>
<tr>
<td>javax.crypto.SecretKeyFactory</td>
<td>convert key into key specification and vice-versa</td>
</tr>
<tr>
<td>javax.crypto.spec.SecretKeySpec</td>
<td>specification of a symmetric secret key</td>
</tr>
<tr>
<td>java.security.SecureRandom</td>
<td>generate secured pseudo-random numbers</td>
</tr>
<tr>
<td>javax.security.Signature</td>
<td>digital signature</td>
</tr>
</tbody>
</table>

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.

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 deriveEncryptionKey (line 1) and then stored in $r0$. So, a misuse could happen if a constant is passed to deriveEncryptionKey in the second parameter.
- Often, developers use a \texttt{String} object to hold the password and then use \texttt{toCharArray()} to convert to the required type (i.e., \texttt{char[]}) when necessary. However, the intention of designing \texttt{PBEKeySpec} constructor to only accept \texttt{char[]} instead of \texttt{String} is to avoid using \texttt{String}, since \texttt{String} object is immutable, therefore, they cannot be destroyed or modified after instantiation until garbage collection reclaims the memory. Given that garbage collection occurs randomly and is out of the control of developers, the password may survive in memory for a long time, increasing the risk of being exploited by attacks.

- 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.

- 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 \texttt{String} object and must be cleared immediately after the use of the \texttt{PBEKeySpec} object. Developers unfortunately often overlook the call to the \texttt{clearPassword()} of \texttt{PBEKeySpec}. In this example code, the method \texttt{clearPassword} in line 15 is correctly called, so there is no misuse.

- The second misuse (\textit{TypestateError} error) also relates to the same API object \texttt{$r2$} and occurs in the same method. It specifies that the misuse is caused by the fact that the \texttt{MessageDigest} object had not been in the state to call method \texttt{reset()}, instead, method \texttt{digest} or \texttt{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\textsubscript{SAST}. Actually CogniCrypt\textsubscript{SAST}'s reports may include \textit{ImpreciseValueExtractionError} notifications, which indicate that CogniCrypt\textsubscript{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\textsuperscript{[16]} a dedicated tool for supporting the extraction of JCA API usage instances.

<table>
<thead>
<tr>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>extit{ConstraintError}</td>
<td>Unrecommended arguments are given. e.g., MD5 as hashing algorithm.</td>
</tr>
<tr>
<td>extit{RequiredPredicateError}</td>
<td>Arguments are not properly created. e.g., constant values are used while values are required to be randomly generated.</td>
</tr>
<tr>
<td>extit{TypestateError}</td>
<td>A JCA API object is not in the right state to invoke a certain method.</td>
</tr>
<tr>
<td>extit{IncompleteOperationError}</td>
<td>Tasks are not completed using JCA API objects. e.g., a MessageDigest object is invoked before passing any information into it via calls to methods \texttt{digest} or \texttt{update}.</td>
</tr>
<tr>
<td>extit{ForbiddenMethodError}</td>
<td>Unrecommended API methods are invoked. e.g., \texttt{PBEKeySpec(char[] password)} is one of the constructor of JCA API \texttt{PBEKeySpec} for deriving cryptographic keys from a given password. Since a key generated without \texttt{salt} has been proven to be weak, this constructor should be used in specific scenarios.</td>
</tr>
<tr>
<td>extit{NeverTypeError}</td>
<td>Certain types are forbidden when storing sensitive information. e.g., password value for \texttt{PBEKeySpec} should never be stored as type \texttt{String}, but \texttt{char[]}. Since object \texttt{String} is immutable in Java, password information in this type cannot be explicitly freed from memory. The garbage collector is in charge of deleting it, yet it is unpredictable from a user standpoint, opening opportunities for password leakage.</td>
</tr>
</tbody>
</table>

Finally, we have implemented a crawler for collecting on Google Play some metadata (e.g., category, rate, etc.) associated to AndroidZoo apps. 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,

Listing 2: CogniCrypt\textsubscript{SAST} Report Example

B. Static API usage checker

We leverage CogniCrypt\textsubscript{SAST}\textsuperscript{[20]}, a static analyzer of the CogniCrypt\textsuperscript{[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\textsuperscript{[20]}, a static analyzer of the\textsuperscript{[22]}. CogniCrypt\textsubscript{SAST} checks JCA APIs against a set of rules that were manually specified by security experts using CrySL\textsuperscript{[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\textsubscript{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\textsuperscript{2} showcases an example of a report generated by CogniCrypt\textsubscript{SAST} where misuses are hierarchically grouped by classes and methods. In this example, 2 misuses are found in method \texttt{java.lang.String} of class \texttt{com.umeng.common.util.h} of app \texttt{com.lovinc.radio}:

- The first misuse, a \textit{ConstraintError} of API \texttt{MessageDigest}, is reported in line 4, with details indicating that argument “MD5” is not recommended when invoking API method \texttt{getInstance(java.lang.String)}.

- The second misuse (\textit{TypestateError} error) also relates to the same API object \texttt{$r2$} and occurs in the same method. It specifies that the misuse is caused by the fact that the \texttt{MessageDigest} object had not been in the state to call method \texttt{reset()}, instead, method \texttt{digest} or \texttt{update} should be invoked before.
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 apps [23], the most diverse dataset available to the research community. Since AndroZoo continuously collects any apps 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. [23] 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.

Eventually, 146,226 apks are excluded from the dataset of apps. 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 no difference between the two distributions, implying that our
The final dataset is still representative of the initial dataset (at least w.r.t app sizes).

Metadata Collection. Some of our investigations require up-to-date 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 CogniCryptSAST 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 CogniCryptSAST 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 CogniCryptSAST 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 CogniCryptSAST 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 issue 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.

\[
\begin{align*}
    m_j & \leftarrow \text{total number of API misuses}(apk_j) \\
    u_j & \leftarrow \text{total number of API usages}(apk_j) \\
    r_j & := \frac{m_j}{u_j}
\end{align*}
\]

We consider the ordered ratio list $R_i$ of app lineage $l_i$ as $R_i := \{r_1, r_2, \ldots, r_n\}$, 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 $s_i = \text{linear regress}(R_i)$ 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](image)

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 CogniCryptSAST. On median
average, 10 JCA API usage instances can be identified per app, while 75% of apps include at least 5 usage instances. Table IV summarizes the statistics on API misuses among the apps that include JCA API usages. 96% of apps 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>
<thead>
<tr>
<th>Types</th>
<th>Numbers Related Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of APKs with misuse</td>
<td>96% number of APKs without misuses is 24,880</td>
</tr>
<tr>
<td>Percentage of app lineages with misuse</td>
<td>97.6% number of lineages without misuses is 942</td>
</tr>
<tr>
<td>Misuse to usage ratio</td>
<td>1 : 5.53 total number of API usages is 23,281,216 and misuses is 4,210,667</td>
</tr>
</tbody>
</table>

We detail the misuse spread in Fig. 4. As shown by the misuse distributions in apps 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 apps. 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.

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 V ranks the JCA APIs based on such a ratio. We provide details of misuse ratio distributions for each API in Fig. 5. The median value of misuse ratio for most of APIs is 0. For these APIs, only IncompleteOperationError and PBEKeySpec have a higher ratio. Based on their median values, ConstraintError has a higher ranking by types. 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 IncompleteOperationError and PBEKeySpec have a higher ratio. Based on their median values, ConstraintError has a higher ranking by types. 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](image)

### Table V: Ranking of Misuses

<table>
<thead>
<tr>
<th>API Method</th>
<th>Type</th>
<th>%</th>
<th>Misuse #</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageDigest.getInstance(java.lang.String)</td>
<td>ConstraintError</td>
<td>60.93%</td>
<td>2,585,892</td>
</tr>
<tr>
<td>Cipher.getInitializationParameterSpec()</td>
<td>IncompleteOperationError</td>
<td>11.64%</td>
<td>490,157</td>
</tr>
<tr>
<td>MessageDigest.getHashValue()</td>
<td>ForbiddenMethodError</td>
<td>0.51%</td>
<td>21,565</td>
</tr>
<tr>
<td>SecretKeySpec.init()</td>
<td>NewTypeError</td>
<td>0.01%</td>
<td>374</td>
</tr>
<tr>
<td>SecretKeySpec.init()</td>
<td>RequiredPredicateError</td>
<td>14.63%</td>
<td>615,921</td>
</tr>
<tr>
<td>SecretKeySpec.init()</td>
<td>TypeCastException</td>
<td>12.27%</td>
<td>516,758</td>
</tr>
</tbody>
</table>

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.

Note: method <init> is the constructor of the class while method init is a common method of the class.
The JCA APIs for implementing cryptography are widely misused across Android apps. Usage mistakes range from issues with parameter initialization to mishaps with the sequence of API method invocations. Nevertheless, all crypto-APIs are not similarly affected by misuse cases.

### TABLE VI: JCA API Misuse to Usage Ratio Ranking

<table>
<thead>
<tr>
<th>API</th>
<th>Misuse Ratio</th>
<th>Misused APK %</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBESecretKeySpec</td>
<td>0.462098</td>
<td>55.50%</td>
</tr>
<tr>
<td>MessageDigest</td>
<td>0.287114</td>
<td>93.02%</td>
</tr>
<tr>
<td>Cipher</td>
<td>0.271808</td>
<td>50.77%</td>
</tr>
<tr>
<td>PBESecretKeySpec</td>
<td>0.251944</td>
<td>97.26%</td>
</tr>
<tr>
<td>Signature</td>
<td>0.204835</td>
<td>23.76%</td>
</tr>
<tr>
<td>KeyPair</td>
<td>0.102394</td>
<td>27.58%</td>
</tr>
<tr>
<td>IvParameterSpec</td>
<td>0.067249</td>
<td>3.05%</td>
</tr>
<tr>
<td>SecretKeySpec</td>
<td>0.058732</td>
<td>27.81%</td>
</tr>
<tr>
<td>Mac</td>
<td>0.0567249</td>
<td>3.05%</td>
</tr>
<tr>
<td>KeyStore</td>
<td>0.0517380</td>
<td>27.74%</td>
</tr>
<tr>
<td>HMACParameterSpec</td>
<td>0.0402394</td>
<td>27.58%</td>
</tr>
<tr>
<td>SecretKeySpec</td>
<td>0.0402394</td>
<td>27.58%</td>
</tr>
<tr>
<td>AlgorithmParameters</td>
<td>0.028322</td>
<td>9.12%</td>
</tr>
<tr>
<td>KeyPairGenerator</td>
<td>0.008141</td>
<td>0.79%</td>
</tr>
<tr>
<td>SecretKeySpec</td>
<td>0.005865</td>
<td>0.60%</td>
</tr>
<tr>
<td>SecretKeySpec</td>
<td>0.003974</td>
<td>0.35%</td>
</tr>
<tr>
<td>SecureRandom</td>
<td>0.0033244</td>
<td>0.94%</td>
</tr>
<tr>
<td>KeyPair</td>
<td>0.000019</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

![Fig. 6: Misuse Ratio Distribution based on APIs](image)

Three JCA APIs (namely, SecretKey, HMACParameterSpec and DSAKeyGenParameterSpec) 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 DHParameterSpec, 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.

**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 V-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 usage 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 usage 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., \(\geq 4.5\)) 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**

<table>
<thead>
<tr>
<th></th>
<th>MF</th>
<th>MI</th>
<th>MFI</th>
<th># Rank</th>
<th>Usage level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td></td>
<td>53,030</td>
<td>50,183</td>
<td>4,483</td>
<td>50.76%</td>
</tr>
<tr>
<td></td>
<td>16,341</td>
<td>72,143</td>
<td>14,486</td>
<td>53.38%</td>
<td></td>
</tr>
<tr>
<td>Reputed</td>
<td></td>
<td>4,200</td>
<td>2,928</td>
<td>440</td>
<td>50.46%</td>
</tr>
<tr>
<td></td>
<td>3,809</td>
<td>2,898</td>
<td>1,982</td>
<td>53.58%</td>
<td></td>
</tr>
<tr>
<td>Finance</td>
<td></td>
<td>179</td>
<td>153</td>
<td>30</td>
<td>50.55%</td>
</tr>
<tr>
<td></td>
<td>2,22</td>
<td>392</td>
<td>195</td>
<td>62.92%</td>
<td></td>
</tr>
</tbody>
</table>

![Fig. 7: Distribution of Slopes (Misuse Trend)](image)

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.

**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 VIII 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 (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.

**TABLE VIII: Misuse Update Ranking by APIs**

<table>
<thead>
<tr>
<th>API</th>
<th>MF # Rank</th>
<th>MI # Rank</th>
<th>MFI # Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cipher</td>
<td>20,574 1</td>
<td>21,751 1</td>
<td>2,459 2</td>
</tr>
<tr>
<td>MessageDigest</td>
<td>17,688 2</td>
<td>20,451 2</td>
<td>1,283 3</td>
</tr>
<tr>
<td>IvParameterSpec</td>
<td>14,300 3</td>
<td>14,150 3</td>
<td>71 7</td>
</tr>
<tr>
<td>SecretKeySpec</td>
<td>14,085 4</td>
<td>14,126 4</td>
<td>71 7</td>
</tr>
<tr>
<td>KeyGenerator</td>
<td>3,372 5</td>
<td>3,465 5</td>
<td>65 8</td>
</tr>
<tr>
<td>Mac</td>
<td>874 6</td>
<td>850 6</td>
<td>9,382 1</td>
</tr>
<tr>
<td>Signature</td>
<td>964 7</td>
<td>516 7</td>
<td>196 4</td>
</tr>
<tr>
<td>KeyStore</td>
<td>580 8</td>
<td>268 8</td>
<td>153 5</td>
</tr>
<tr>
<td>PBEKeySpec</td>
<td>155 9</td>
<td>304 9</td>
<td>83 6</td>
</tr>
<tr>
<td>SecureRandom</td>
<td>134 10</td>
<td>180 11</td>
<td>0 12</td>
</tr>
<tr>
<td>PBEParameterSpec</td>
<td>122 11</td>
<td>212 10</td>
<td>0 12</td>
</tr>
<tr>
<td>AlgorithmParameters</td>
<td>47 12</td>
<td>22 12</td>
<td>2 11</td>
</tr>
<tr>
<td>KeyPairGenerator</td>
<td>14 13</td>
<td>20 13</td>
<td>0 12</td>
</tr>
<tr>
<td>GCMPParameterSpec</td>
<td>13 14</td>
<td>13 14</td>
<td>0 12</td>
</tr>
<tr>
<td>SecretKeyFactory</td>
<td>1 15</td>
<td>5 15</td>
<td>17 10</td>
</tr>
<tr>
<td>KeyPair</td>
<td>0 16</td>
<td>0 16</td>
<td>0 12</td>
</tr>
</tbody>
</table>

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.

![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.](image)
both RequiredPredicateError and ConstraintError generally concern the initialization or the selection of proper arguments to API methods.

### TABLE IX: Misuse Update Ranking by Types

<table>
<thead>
<tr>
<th>API Method</th>
<th>MF % Rank</th>
<th>MI % Rank</th>
<th>MFI % Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequiredPredicateError</td>
<td>66.78%</td>
<td>69.99%</td>
<td>4.86%</td>
</tr>
<tr>
<td>ConstraintError</td>
<td>29.10%</td>
<td>23.72%</td>
<td>7.08%</td>
</tr>
<tr>
<td>TypestateError</td>
<td>3.35%</td>
<td>3.38%</td>
<td>12.07%</td>
</tr>
<tr>
<td>IncompleteOperationError</td>
<td>0.71%</td>
<td>0.84%</td>
<td>76.67%</td>
</tr>
<tr>
<td>NeverTypeOfError</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.05%</td>
</tr>
<tr>
<td>ForbiddenMethodError</td>
<td>0.02%</td>
<td>0.01%</td>
<td>0.27%</td>
</tr>
</tbody>
</table>

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 as 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

<table>
<thead>
<tr>
<th>API Method</th>
<th>From</th>
<th>To</th>
<th>%</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac.init()</td>
<td>update</td>
<td>doFinal</td>
<td>9.41%</td>
<td>expected method call of init was added during update but still require further method call of update or doFinal which is missing</td>
</tr>
<tr>
<td>Mac.init()</td>
<td>doFinal</td>
<td>update</td>
<td>4.58%</td>
<td>before update, new was call but missing method call of update or doFinal. However, update even removed method call of init</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>API Method</th>
<th>MF % Rank</th>
<th>MI % Rank</th>
<th>MFI % Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageDigest.getInstance(java.lang.String)</td>
<td>19.183</td>
<td>25.13%</td>
<td>1</td>
</tr>
<tr>
<td>Reason: parameter with value: MDS, SHA-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>143.070</td>
<td>18.43%</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Reason: parameter is not randomized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.681</td>
<td>19.92%</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Reason: first parameter is not randomized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.559</td>
<td>8.59%</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Reason: second parameter is not properly generated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.445</td>
<td>6.82%</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Reason: second parameter is not properly randomized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>366</td>
<td>0.43%</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Reason: expected to call method init before</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>391</td>
<td>0.54%</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Reason: private key (first parameter) is not properly generated</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 misuses 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
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 CogniCryptSAST. We have
actually benchmarked a set of apps to check the false positive
rates of CogniCryptSAST. Our manual verification confirms
that CogniCryptSAST is effective. We have only observed
one case where CogniCryptSAST 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 CogniCryptSAST 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 several different angles, we did not exhaust
all possible ways. Therefore, there are still chances for the
assumption to be true 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 CogniCryptSAST to detect
misuses of crypto-APIs in Android apps. To the best of our
knowledge, CogniCryptSAST 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 CogniCryptSAST. 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, Chatzikostantinou 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 lever-
age 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 [30] also
investigate the evolution of security and privacy issues in
Android apps. These two approaches, although conducted
separately, both show 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


