Janovsky, A., Maiorca, D., Macko, D., Matyas, V., Giacinto, G. (2023). *Explaining the Use of Cryptographic API in Android Malware*. In: Van Sinderen, M., Wijnhoven, F., Hammoudi, S., Samarati, P., Vimercati, S.D.C.d. (eds) E-Business and Telecommunications. ICSBT SECRYPT 2022 2022. Communications in Computer and Information Science, vol 1849. Springer, Cham

The publisher's version is available at:

https://doi.org/10.1007/978-3-031-45137-9_4

Explaining the use of Cryptographic API in Android Malware

Adam Janovsky[1], Davide Maiorca[2], Dominik Macko[1], and Vashek Matyas[1] and Giorgio Giacinto[2]

[1] Masaryk University, Czech Republic
adamjanovsky@mail.muni.cz
[2] University of Cagliari, Italy
{davide.maiorca,giacinto}@unica.it

Abstract. Cryptography allows for guaranteeing secure communications, concealing critical data from reverse engineering, or ensuring mobile users' privacy. Android malware developers extensively leveraged cryptographic libraries to obfuscate and hide malicious behavior. Various system-based and third-party libraries provide cryptographic functionalities for Android, and their use and misuse by application developers have already been documented. This paper analyzes the use of cryptographic APIs in Android malware by comparing them to benign Android applications. In particular, Android applications released between 2012 and 2020 have been analyzed, and more than 1 million cryptographic API expressions have been gathered. We created a processing pipeline to produce a report to reveal trends and insights on how and why cryptography is employed in Android malware. Results showed that the usage of cryptographic APIs in malware differs from that made in benign applications. The different patterns in the use of cryptographic APIs in malware and benign applications have been further analyzed through the explanations of Android malware detectors based on machine learning approaches, showing how crypto-related features can improve detection performances. We observed that the transition to more robust cryptographic techniques is slower in Android malware than in benign applications.

Keywords: Cryptography · Android · Malware

1 INTRODUCTION

The increased number of Android operating system users during the last decade, reaching almost 3 billion in 2021 [7], is one of the key motivations of the increase of security threats targeting Android showed [27]. The use of Smartphones to store an increasing number of personal and business-related information, including health, finance, and access tokens, made it one of the targets of cyber attacks. Cryptographic primitives are employed to conceal critical information and securely carry out communication with internal components, applications, and web services. At the same time, it is natural to imagine malware authors leveraging cryptography in many artful ways to serve their malevolent objectives.

For instance, cryptography equips attackers with the ability to fingerprint the parameters of an infected device, encrypt users' media files, establish a secure connection with a command-and-control server, or manage ransom payments carried out by victims infected by, e.g., ransomware.

Previous research conveyed a significant effort in analyzing cryptography in benign applications. The focus was mainly on the misuse of cryptographic application programming interface (API) in benign Android applications, i.e., finding and eliminating vulnerabilities in the employed crypto-routines that may allow attackers to obtain sensitive information [11,31,8,39].

To the best of our knowledge, however, no study explored how cryptography is currently employed in *malicious* applications, the only example being a previous work from the authors of this contribution [18]. That paper was focused on providing details on the cryptographic API used in Android malware, its evolution over time, and the potential contribution that such a study can provide for improving malware detectors.

Notably, this paper is an extended version of a conference paper published in SECRYPT 2022 [18]. In this extended work, we aim to make a step forward by focusing on machine-learning techniques for Android malware detection. In particular, we aim at assessing the effectiveness of considering the information on cryptographic API as additional features in the design of Android malware detectors.

We can summarise the aim of this paper as the answer to the following research questions related to cryptography and Android malware:

1. **RQ.1:** Are there significant differences in how cryptography is employed in benign and malicious applications?
2. **RQ.2:** How do features related to cryptography affect the performances of Android malware detectors?

We believe that answering these questions will shed more light on the mechanisms of Android malware, providing new insights for its analysis, characterization, and detection. To this end, in this paper, we propose two main contributions. First, we deliver a comprehensive comparison of how cryptography is employed in 603 937 malicious and benign applications released in the last decade. Such a comparison is carried out with an open-source[3], scalable approach that inspects (among others) the usage of hash functions, symmetric and public-key encryption, PRNGs, etc. In total, we inspect over 10^6 of cryptographic API expressions.

Second, we show that cryptographic features demonstrate their discriminant power in distinguishing malicious and benign applications by employing techniques inherited from the interpretation of learning models. This allows us to point out possible connections between cryptographic API and malicious actions and augment state-of-the-art malware detectors' performances.

The attained results show many intriguing and surprising trends. For example, unlike benign applications, malware authors do not typically resort to strong cryptography to perform their actions. We show that malware often favors the use

[3] The code is accessible from `github.com/adamjanovsky/AndroidMalwareCrypto`.

of cryptographically defeated primitives, e.g., weak hash functions MD5 [43] or SHA-1 [41], or symmetric encryption scheme DES [6]. These insights can also be especially useful to learning-based models, which can leverage these cryptographic trends to improve the detection rate of malware. We believe the results presented in this work can constitute a seminal step to foster additional research on the relationship between cryptography and Android malware.

The paper is organized as follows: Section 2 provides all the necessary technical background. The proposed methodologies are reported in Sections 3 and 4, where the processing pipeline and the machine-learning approaches are described, respectively. Reported results on the statistics of crypto usage and in-depth analysis for various crypto functions are reported in Section 6. Section 7 shows how explaining machine learning-based Android malware detectors can provide useful information for threat analysis. Moreover, crypto-related features can be essential in detecting some malware samples. The limitations of the present study are discussed in 8 while Section 9 provides a thorough analysis of the related research works. Conclusions are reported in Section 10.

2 TECHNICAL BACKGROUND

This section provides the basic technical elements that will be used in the rest of the paper. We first describe the structure of Android applications. Then, we provide an overview of the techniques used to analyze Android applications. Finally, we describe the prominent functionalities of the cryptographic APIs that can be employed in Android applications.

2.1 Background on Android

Android applications can be represented as zipped `.apk` (Android application package - APK) archives comprising: *(i)* The `AndroidManifest.xml` file, which provides the application package name, the name of the app basic components, and the permissions that are required for specific operations; *(ii)* One or more `classes.dex` files, which represent the application executable(s), and which contain all the implemented classes and methods executed by the app. This file can be disassembled to a simplified format called `smali`; *(iii)* Various `.xml` files that characterize the application layout; *(iv)* External resources that include images and native libraries.

Although Android applications are typically written in `Java`, they are compiled to an intermediate bytecode format called `Dalvik`, whose instructions are contained in the `classes.dex` file. This file is parsed at install time and converted to a native ARM code executed by the Android RunTime (ART). The use of ART allows speeding up the execution in comparison to the previous runtime (`dalvikvm`, available till Android 4.4), where applications were executed with a just-in-time approach: during installation, the `classes.dex` file was only slightly optimized, but not converted to native code.

2.2 Analysis Techniques for Android applications

Android applications, like any other application written for different platforms, can be analyzed either statically or dynamically. Static analysis can be performed in two different ways. The Dalvik bytecode's instructions are disassembled, or the executable is decompiled to its Java source. Typical analysis techniques involve, among others, program slicing, data flow and taint analysis and the extraction of the application call graphs. Dynamic analysis can be performed by tracing the execution of the instructions, as well as the changes in memory during the execution of the applications.

Both approaches feature their limitations, and a comprehensive analysis requires the exploitation of the complementarities. Static analysis can be evaded by obfuscation techniques, such as renaming user-implemented functions, modifying the call graph, and using reflection or encryption API [25,16]. Dynamic analysis can be especially challenging due to the so-called *path-explosion* problem, where the application should be stimulated to take different execution branches. This operation is particularly complex in Android apps, as there are numerous ways to interact with them. As applications are typically executed in emulated environments, a malicious application can first check whether the application is being debugged or not. Finally, dynamic analysis can be resource- and time-consuming.

The choice of the right technique explicitly depends on the analysis goals. Static analysis is typically recommended for large-scale analyses, as it is much faster to carry out, and its scalability outweighs its limitations over many samples.

2.3 Cryptography in Android

Android developers typically have several means to implement cryptographic functionality for their applications: *(i)* Using Java Cryptographic Architecture (JCA) via Android API; *(ii)* Using third-party Java cryptographic libraries; *(iii)* Using third-party native cryptographic libraries; *(iv)* Designing and/or employing their cryptographic functions. Note that the last method is widely discouraged by the cryptographic community and is unfeasible to be reliably employed, as it has no well-grounded fingerprint. We also stress that, in most cases, developers do not need to develop their cryptographic functions because they can more easily and reliably employ readily available tools. For these reasons, we will not discuss the *(iv)* case in the rest of the section.

Android API cryptographic functionalities are delivered via JCA. JCA provides a stable set of classes and functions that can be called from two main packages, `javax.crypto` and `java.security` [35,14]. These packages contain more than 100 classes covering the majority of cryptographic primitives and protocols, such as hash functions, symmetric encryption schemes, digital signature algorithms, and so forth. Although the Android documentation explicitly recommends using specific primitives[4], many weak and insecure cryptographic

[4] AES-256 in CBC or GCM mode, SHA-2 for hash functions, SHA-2 HMAC for MACs and SHA-2 ECDSA for signatures as of early 2020 [14].

primitives (such as the MD5 hashing function or the symmetric cipher DES) can be chosen. Apart from providing the API, the JCA introduces an abstraction layer of the so-called Cryptographic Service Providers. Such providers register themselves at the JCA and are responsible for implementing any subset of the API. Different Android versions suggest using different providers, such as `BouncyCastle` [21] and `Conscrypt` [15], which are among the most popular ones. While these providers differ in their internals, they must comply with the API and expose identical function names and argument ranges.

Apart from the Android API, practically any cryptographic functionality can be supplied by some third-party library. Still, there is no curated list of either Java or native cryptographic libraries for the Android platform to our best knowledge. During our study, we noticed libraries specifically focusing on a subset of cryptographic functions, such as providing functionality only for AES encryption. We also noticed libraries exposing various cryptographic primitives and protocols, such as the OpenSSL library. These full-fledged libraries can provide functionalities similar to the Android API.

3 METHODOLOGY

This section describes the methodology employed to extract and analyze the cryptographic API embedded in Android applications. We start by formalizing the problem and properly defining its domain and constraints. We then show how we implemented this formalism by discussing our developed analysis framework. Our findings are based on the static analysis of the Java source code obtained by decompiling the Android executables.

3.1 Problem Formalization

We organize the problem formalization in two parts: part one treats the definition of the crypto-routines of interest for our analysis, and part two describes the process of locating those routines in the application source code.

I. Definition of Crypto-Routines. Given a set of Android applications, we denote the set of all possible functions $\mathbb{F}$ contained in their source code as:

$$\mathbb{F} = \mathbb{U} \cup \mathbb{S} \cup \mathbb{T} = \mathbb{C} \cup \mathbb{C}^c,$$

Where $\mathbb{U}$ represents the set of functions defined by the user, $\mathbb{S}$ is the set of system-related functions contained in the Android SDK, and $\mathbb{T}$ is the set of functions belonging to third-party libraries. Given a set of known crypto-related functions $\mathbb{C}$, our goal is to study the intersection of $\mathbb{C}$ and $\mathbb{S}$, denoted as $\mathbb{F}_{cs}$. In other words, $\mathbb{F}_{cs}$ is the set of cryptography-related functions that are defined in the system package (in Android, represented by JCA functions). In this analysis, we discard custom cryptographic functions that users or third parties may implement. The automatic detection of such functions would be a complex task in a large-scale analysis, which may lead to false positives (or negatives) without further manual inspection. In our study, we solely aim to answer what functions from $\mathbb{F}_{cs}$ the malware authors favor.

From the cryptographical perspective, the functions contained in $\mathbb{F}_{cs}$ can be divided into the following categories: *(i) Hash functions.* Cryptographic hash functions such as MD5, SHA-1, or SHA-2; *(ii) Symmetric encryption.* Symmetric cipher primitives such as AES, DES, or RC4; *(iii) Public-key encryption.* Asymmetric primitives, in Android represented by the RSA cryptosystem; *(iv) Digital signature algorithms.* Primitives that empower digital signatures, e.g., ECDSA; *(v) MAC algorithms.* Primitives that construct Message Authentication Codes, also called MACs; *(vi) PRNG.* Functions to run pseudo-random number generators (PRNG); *(vii) Key agreement protocols.* Algorithms for key exchange, in JCA represented by Diffie-Hellman protocol; *(viii) Others.* Functions that do not fall into any of the previous categories.

II. Locating Cryptographic API. All functions in $\mathbb{F}_{cs}$ are available through two Java packages in Android API: `javax.crypto` and `java.security`. Our research goal is to reveal *which cryptographic functions have been chosen and directly employed by the authors.* Notably, Android applications typically contain third-party packages that invoke crypto functions. We aim to exclude those packages from our analysis as the application authors did not contribute to them.

Thus, for each Android sample, we are interested in extracting the cryptographic API $\mathbb{F}_a \subseteq \mathbb{F}_{cs}$ that is invoked from user-defined functions $\mathbb{U}$. To obtain the functions belonging to $\mathbb{F}_a$, we developed the following two-steps procedure: *(i)* We automatically detect the classes that belong to third-party or system libraries, and we exclude them from the set of classes that should be explored. By doing so, we establish the list of *user-implemented functions* $\mathbb{U}$; *(ii)* We extract all references to crypto-related functions $\mathbb{F}_{cs}$ that are invoked directly from $\mathbb{U}$.

The first step is motivated by the discovery [42] that more than 60% of Android APK[5] code (on average) originates from third-party packages. To study user-authored code, it is therefore critical to differentiate, with reasonable certainty, whether a class belongs to a third-party library or not. This task can be extremely challenging and was extensively studied, e.g., by [42,24,4]. It does not suffice to merely search for the `import` clauses in the decompiled source code since non-system packages could be renamed. This scenario is especially frequent in malicious applications, as the authors aim to defend against forensic investigation techniques. Inspired by the systematic review of third-party package detectors [44], we opted to tackle this task with `LibRadar`, a popular third-party library detection tool that uses clustering techniques and complex signatures to recognize such libraries [24]. The results reported in the review paper show that `LibRadar` achieved the highest precision and second-highest recall while it took approx. 5 seconds to evaluate an APK on average. The runner-up requires over 80 seconds per APK, which would be unsuitable for large-scale analysis. `LibRadar` was trained on a large dataset of Android applications and can reliably fingerprint more than 29 000 third-party libraries without relying on package names. Consequently, `LibRadar` can

[5] Android Application Package, an archive that encapsulates the whole Android application.

identify obfuscated packages. Using `LibRadar`[6], we filter the identified third-party packages of an APK from subsequent cryptographic API analysis.

3.2 Crypto API Extraction Pipeline

Given a dataset containing Android APKs, our system generates a comprehensive report of the embedded cryptographic API. Our system requires configuration files for the to-be-conducted experiment. Apart from other choices, the files contain a list of APKs that can be loaded from a disk or downloaded from the Internet.

The APKs are then processed in parallel, and each sample traverses the following pipeline:

1. **Pre-processor**. This module decompiles the APKs to obtain their Java source code. Then, the third-party packages of the APKs are identified, and the whole Java source code of the APKs is extracted.
2. **Crypto-extractor**. This module extracts and analyzes the cryptographic function call sites in the application source code. Their filtering is achieved by matching pre-defined regular expressions. Additionally, the crypto-extractor also detects both Java and native third-party cryptographic libraries.
3. **Evaluator**. This module stores, organizes, and aggregates the information retrieved by the analyzed APKs to a JSON record.

The evaluator outputs a report of the cryptographic usage for each APK. We designed the system in a modular fashion to allow for the addition of other modules for extracting further valuable insights from the APKs.

4 Cryptography and Machine Learning

To accurately detect malicious applications through machine learning approaches, multiple features are typically extracted. Among these features, cryptographic usage statistics are undoubtedly helpful in pointing out differences between benign and malicious applications. In this paper, we also aim to explore *whether the statistics on the usage of cryptographic functions can be useful to recognize malicious samples from benign ones effectively.* To answer this question, we propose three approaches that employ machine learning techniques, described in the following.

4.1 Cryptographic Learning Model

The first technique consists of defining a learning-based system whose structure is inspired by other popular detection systems [10,9,26]. In particular, the proposed

[6] Since `LibRadar` requires a large Redis database to run (preventing parallelization), we actually leveraged its lightweight version `LiteRadar`. Before doing so, we compared the output of both tools on a small subset to find out that this decision has a negligible effect on the number of detected libraries.

system performs the following steps: *(i)* it takes as an input an Android application and extracts its cryptographic API usage with the pipeline described in Section 3.2; *(ii)* it encodes this statistics into a vector of *features*; *(iii)* it trains a machine-learning classifier to predict a benign/malicious label.

The feature vector includes features that can be categorized into the following three sets:

- **Set A**: flags indicating the use of third-party cryptographic libraries (both Java and native).
- **Set B**: frequencies of specific cryptographic API constructors and imports of crypto-related classes, e.g., number of DES constructors in a sample.
- **Set C**: aggregated statistics of call sites and imports related to categories of cryptographic primitives: hash functions, symmetric encryption schemes, and so forth. For example: how many distinct hash functions a sample uses.

By joining these sets, we obtain 300 potentially informative features. These features are further filtered with the following feature selection process. The dataset with candidate features is split in a 9:1 ratio into training/test sets. First, we examine all possible pairs of features. If a pair exhibits Pearson's correlation coefficient higher than 0.95, we drop a random feature of such a pair. Second, we remove the features deemed uninformative by Boruta [20]. Boruta is a supervised algorithm that iteratively replicates features, randomly permutates their values, trains a random forest, and removes redundant features based on the z-score. This feature selection process yields 189 features on the dataset used in this study, whose details are reported in Section 6.1.

To choose the classifier best suited for our study, we resorted to a preliminary experiment on a dataset similar to the one used in this study. The following machine learning approaches have been considered: Naive Bayes, Logistic Regression, Support Vector Machines with linear kernel, Random Forest, Gradient Boosted Decision Trees (GBDT), and Multilayer Perceptron (MLP). The classifiers' hyperparameters have been tuned using 10-fold cross-validation on the training dataset, optimizing for the F1 score. The best classifier, according to the performances on the F1 score, was selected as the candidate model for carrying out the explanation analysis, followed by experiments aimed at showing the capability of crypto-related features to enhance the performances of Android malware detectors. The selected classifier turned out to be the Random Forest (which works as a majority-voting ensemble of decision trees trained on different subsets of the data) with an F1 score of 57.07%. In this preliminary study, GBDT and MLP had an F1 score of 56.89% and 56.41%, respectively, while the other classifiers provided significantly smaller performances.

4.2 Explaining the Learning Model

To further advance the understanding of cryptographic API in Android detection, we extracted *explainations* from the predictions of the cryptographic classifier. Explanation techniques allow understanding of the learning process results through the analysis of the training samples' features that influence the classifiers' decisions. In this paper, we used both *global* and *local* feature importances as reported in previous work dealing with the explanation of Android malware detectors [28,29]. The global analysis evaluates the impact of the features averaged over the whole dataset, while the local analysis evaluates the impact of the features on specific samples.

To interpret the classifier's predictions, we used Shapley additive explanations (SHAP) [22] that are successfully used outside the computer science field. SHAP can consistently explain both local predictions and global feature importance by measuring each feature's contribution to the prediction. This method uses Shapley values [38] from coalitional game theory. Each player is a feature or a coalition of features, and the game (payout) is the prediction. Shapley values are considered optimal because they satisfy the properties of efficiency, symmetry, and additivity.

4.3 Enhancing existing malware classifiers

The third approach consists of taking a well-established malware classifier for Android as a baseline and measuring its performance when enhanced with features related exclusively to cryptographic API. To this end, we chose `R-PackDroid` [26], an available learning-based classifier (trained on random forests) based on static features, and we expand its feature set by adding the cryptographic features described above. There are multiple reasons for which this system was chosen as a baseline: (i) It was initially designed to detect ransomware; (ii) It harvests a relatively small number of features; (iii) It features a high detection rate (the original paper documents over 97% F1 score).

Considering the characteristics described above, it would normally be challenging to improve the already strong performance of the system by adding more features. To keep some space for improvement when enhancing the classifier with cryptographic features, we decreased the number of `R-PackDroid` features from 211 to 10 in a controlled manner, leading to an F1 score of 76%. To measure the effect of cryptographic API features on the model, we replicated the following procedure 1000 times: *(i)* We sample 10 random features from `R-PackDroid`[7]; *(ii)* We build a random forest classifier and measure its F1 score; *(iii)* We enhance the 10 `R-PackDroid` features with all 189 cryptographic API features chosen by feature selection; *(iv)* We use the expanded feature set to build another random forest classifier and measure its performance gain over the baseline classifier.

With the three strategies described above, we unveil the role of cryptographic API for malware detection, as will be shown in Section 7.

[7] The tuples were sampled in advance to avoid repetition.

5 IMPLEMENTATION OF THE PROCESSING PIPELINE

We now provide a more detailed description of each module, as well as the technical challenges that we had to face during the development of our system. Our tool is implemented in Python 3.8 and is provided as an open-source repository for further collaboration.

5.1 Pre-processor

The first task of the pre-processor is to obtain the decompiled Java source code of the input APKs. To get object-oriented access to the source code, we instrumented the open-source tool `Androguard` [3]. As `Androguard` supports multiple decompilers, we made preliminary tests with various decompilers to verify that applications would be correctly parsed. The attained results showed that `JADX` decompiler [40] is capable of the most mature recovery of the Java code, and hence was used in this study. For each APK, all `.java` classes in its `.dex` files were recovered. In some samples, a small fraction ($< 1\%$) of classes did not survive the decompilation process. These classes were ignored from further processing.

The second task of the pre-processor is to craft a list of third-party packages residing in the scrutinized APK. As mentioned in the previous section, this task is handled with the help of `LiteRadar` that was trained on a large dataset of Android applications, and can reliably fingerprint more than 29 000 third-party libraries. It should be stressed that the `LiteRadar` does not rely on package names, and can thus identify obfuscated packages as well. Using `LiteRadar`, we check every APK for the presence of third-party packages. Each package identified as a third party is then excluded from the cryptographic API analysis.

5.2 Crypto-extractor

The crypto-extractor component executes two sub-tasks. The first objective is to gather a list of third-party *cryptographic* libraries imported from the APK. For this scenario, we discriminate between *(i)* Java cryptographic libraries and *(ii)* native cryptographic libraries. In total, we searched for 23 distinct libraries. Their names, together with the reasons behind choosing them, can be found in Section 6.

The candidate list of native cryptographic libraries is then matched inside any of the following three import statements that load a native library directly from the source code: `ReLinker.loadLibrary`, `System.loadLibrary`, and `Native.loadLibrary`. The candidate list of Java cryptographic libraries is compared with the list of third-party packages identified earlier by `LiteRadar`. If any package appears in both lists, we note down the usage of the respective cryptographic library.

The second goal of the crypto-extractor is to collect comprehensive data about cryptographic API usage. Although the packages `javax.crypto` and `java.security` contain more than 100 classes and interfaces, only some of those can reveal insights about the diversity of cryptography usage in the malware. We analyzed all classes in these packages and discarded out-of-scope instances to obtain 86 classes for our analysis. Most of the diversity in the cryptographic API landscape can be explained by the study of *object constructors*, and of their parameters. Whatever the developers' aim concerning cryptography is, they must first create a suitable object to address it. To give an example, when developers want to hash a file, they must first obtain the hash object by calling the constructor `MessageDigest.getInstance()`. When this constructor is called, e.g., with a string parameter `"SHA-256"`, this reveals the probable usage of the SHA-256 hash function in the APK. We specified 333 constructors and parameters and recorded all occurrences of these in the source code[8]. Specifically, we performed a line-by-line search of each of the user-defined classes. If the searched line contained any of the constructors, we note down its usage. By doing so, we collected a rough landscape of cryptography usage in the whole source code. This data was further refined and processed to draw conclusions.

Notably, constructors parameters can be obfuscated (and thus missed by our analysis - e.g., `MessageDigest.getInstance(a)`, where `a` is some variable). In this case, our system cannot properly parse such constructors. However, we argue that this limitation could only partially be statically solved, as even more advanced techniques (such as program slicing) can be easily defeated by more advanced obfuscation [25,16]. Moreover, in large-scale scenarios, the problems introduced by the presence of some obfuscation are significantly outweighed by the dataset size. Nevertheless, for the sake of a fair analysis, we computed the exact fraction of obfuscated constructors for each of the cryptographic primitives we analyzed.

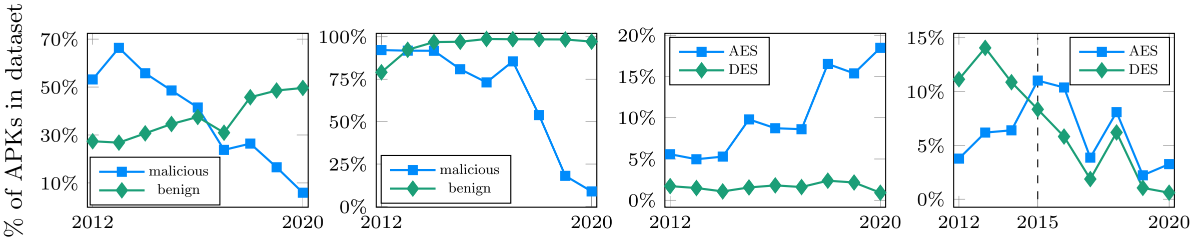
6 TRENDS IN CRYPTOGRAPHY

In this section, we answer RQ.1: *Are there significant differences in how cryptography is employed in benign and malicious applications?* We report the most significant statistics we obtained with the API extraction methodology presented in Section 3.2. First, we discuss the general prevalence of cryptographic API that can be extracted with static analysis by discussing the effects of obfuscation, the incidence of third-party packages, and the overall differences between benign and malicious applications. Then, we provide a more detailed focus on the distribution of cryptographic API in malicious applications.

6.1 Dataset

To gain an all-around view of the cryptographic API landscape in Android applications, we leverage the Androzoo dataset [1]. Currently, Androzoo is the

[8] While having the capability to capture such diverse landscape, in Section 6 we present results only for 220 constructor variants from 8 classes, since the rest is used very rarely. No conclusions can be drawn from such rare events.



(a) cryptographic API (b) third-party package (c) Benign AES/DES (d) Malicious AES/DES

Fig. 1: Time evolution of important dataset characteristics. The y-axis shows the percentage of APKs exhibiting a given feature, whereas the x-axis represents the time in years. In subplot (a) we see the ratio of APKs in which we detected *any* cryptographic API. Subplot (b) shows APKs for which we detect any third-party package. These are the main artifacts of increasing obfuscation. Subplots (c), (d) document the usage of AES vs. DES in benign, and malicious samples respectively. AES has been the most prevailing cipher suite in benign applications since 2012. On the contrary, DES was more popular in malware in previous years, only in 2015 being outrun by AES [18].

largest available dataset of Android applications, containing more than 15 million of APKs. We sampled 302 039 benign applications and 301 898 malicious applications from Androzoo released in the years 2012-2020. We strived for uniform distribution of samples in the studied timeline. Yet, for years 2018 and 2020 we could only collect a limited number of malicious samples – 19 305 and 10 039, respectively. To speed up the computation, we only gathered APKs smaller than 20 MB (approximately 89% of malicious APKs in the Androzoo fulfill this criterion).

To accurately discriminate malicious files, we consider an APK as malicious if it was flagged malicious by at least five antivirus scanners from the VirusTotal service[9], which should reliably eliminate benign files deemed malicious, as reported by Salem [37]. Our samples are predominantly originating from 3 distinct sources: Google Play (60%), Anzhi (19%), and Appchina (13%). Note that the samples were deduplicated on a per-market basis [1] to avoid over-counting.

6.2 Evaluator and post-processing

Once the overall JSON report for all APKs in the dataset is acquired according to the pipeline described in Section 5, we post-process the results by automatically generating CSV files containing usage statistics and the related plots for all categories (and their sub-categories) described in Section 3.1. Apart from that, useful general information about the nature of the dataset is provided in the

[9] `virustotal.com`. The number of VirusTotal positive flags is already contained in the Androzoo dataset.

report. The resulting data serves as input for the pipeline of a cryptographic learning model.

6.3 System deployment

The parallel processing of all 604 thousand samples took 16 days on 42 cores of Intel Xeon X7560, and each core consumed approximately 1.6 GB of RAM. That is the equivalent of processing 7 thousand APKs per 24 hours on a single CPU with 4 cores and 16 GB of RAM, making the system well scalable. The subsequent post-processing of the JSON record to the form of the cryptography usage report takes 5 minutes on a regular laptop with 4 cores.

6.4 General API Distribution

Application Obfuscation The first interesting trend of this study is a decreasing ratio of malicious applications for which we detect usage of cryptographic API, as depicted in Figure 1a. However, such a ratio is not visible for benign applications. We conjecture that this drop does not represent a genuine decrease in usage of cryptographic functionality in time but rather a consequence of an increasing ratio of obfuscated malicious applications. To confirm this hypothesis, we randomly sampled 4444 malicious applications from 2018-2020 that allegedly contained no cryptographic API or third-party libraries. We dissected them using commercial dynamic analysis tool `apklab`[10] and searched for clues of obfuscation. We identified that 98% applications use some form of Android packer, with `jiagu` being the most popular. Each packed application also uses reflection API and dynamic library loading, which prevent static analysis from registering cryptographic API call sites. We also report that 83% of applications use some form of encryption API (AES being the most prevalent, followed by RSA), often to decrypt application resources. Such reduced prevalence of crypto API constitutes a limitation of our study, further discussed in Section 8.

Third-party Packages and Crypto Libraries Another closely related trend is the dropping ratio of third-party packages captured by `LiteRadar` in malicious applications. Before 2018, we documented a high ratio of malware employing third-party packages (86%). Starting with 2018, this ratio quickly drops as depicted in Figure 1b. Similar to obfuscation, this drop is not evident in benign applications. Overall, `LiteRadar` was able to identify at least one third-party package in 94.6% of analyzed goodware with little variance between years (see Figure 1b). On average, 8 packages were identified in benign APKs. This underpins the importance of robust third-party package detection. In contrast to prior work that did not consider third-party package filtering, we discarded over 4 million third-party packages with at least 44 thousand unique package names from the analysis.

[10] Kindly provided by Avast, available at `http://apklab.io`.

3rd-party cryptographic libraries
Java
`whispersystems/curve25519`
`guardianproject/netcipher`
`springframework/security/crypto`
`gnu/crypto`
`apache/shiro/crypto`
`rsa/crypto`
`keyczar`
`jasypt`
`googlecode/gwt/crypto`
`sqlcipher`
`spongycastle`
`bouncycastle`
`facebook/crypt`
native
`crypto-algorithms`
`libgcrypt`
`monocypher`
`PolarSSL`
`tint-AES-C`
`xxHash`
`libsodium`
`openssl`
`libressl`
`wolfssl`

Table 1: List of 23 third-party cryptographic libraries that were searched in each of the studied samples. The Java libraries were identified using the `LiteRadar` tool, whereas the native libraries were matched as case-insensitive regular expressions inside the import statements `ReLinker.loadLibrary`, `System.loadLibrary`, `Native.loadLibrary` from the decompiled source code.

Apart from Android API, cryptographic functions can also be delivered by third-party libraries, typically adopted to integrate functionality missing in system-based libraries. To our best knowledge, no curated list of third-party cryptographic libraries for Android exists. We manually selected 13 Java and 10 native candidate libraries to be searched for. These candidates were found through the Google search engine and in popular databases [5], and their fit was confirmed by manual inspection. Although this process is inherently incomplete, and some libraries could have been missed out, we argue that this is not a practical limitation since the most prevalent libraries were unlikely to be missed, and even these are rarely used. The full list of third-party cryptographic libraries searched in the samples can be found in Table 1.

We identified only 796 benign and 198 malicious applications that import third-party cryptographic libraries. Of the studied libraries, `sqlcipher` was most popular in goodware (found in 622 samples), and `keyczar` was most popular in malware (found in 124 samples). The ratio of these applications has been stable throughout the studied timeline. As for the native libraries, not a single call to a native cryptographic library was detected in the malicious dataset, and merely 91 imports of `OpenSSL` occurred in the benign dataset.

From these results, it is possible to observe that third-party cryptographic libraries are not widely used in Android applications. This aspect demonstrates that attackers often resort to standard crypto functionalities provided by system libraries (that can use various backends, e.g., BouncyCastle).

Crypto API in Goodware and Malware

We now describe the general prevalence of cryptographic API in the dataset presented in Section 6.1 by showing the differences between malicious and benign applications and comparing our results with two studies conducted on benign datasets. For this comparison, we employed: *(i)* A dataset collected in 2012 as a part of the study CryptoLint [11] that we refer to as CryptoLint-B12; *(ii)* A dataset collected in 2016 as a part of Binsight study [31] that we refer to as Binsight-B16. To avoid temporal data drift, we cast four subsets of our dataset: Androzoo-B12, Androzoo-M12, Androzoo-B16, and Androzoo-M16, limited to malicious (M), and benign (B) samples from years 2012 (12), and 2016 (16). As explained in Section 3, our goal is to analyze only cryptographic APIs contained in user-defined code. From this respect, both [11,31] employ weaker methodologies to filter third-party libraries, relying on whitelisting and package names. Conversely, our approach of `LiteRadar` filtering captures the code written by the application authors more reliably. The numbers drawn from the Androzoo datasets serve as conservative estimates, with the real number of cryptographic API call sites even higher. The overall comparison with benign datasets is depicted in Table 2 (also reported in [18]). It can be seen that the malicious datasets have a dramatically higher density of cryptographic API call sites than their benign counterparts.

Dataset	#APKs	#User-def. call sites	#User-def. call sites/10k samples
CryptoLint-B12	145 095	20 967	1445
BinSight-B16	115 683	78 163	7006
Androzoo-B12	39 838	81 698	20 507
Androzoo-B16	37 493	124 705	33 260
Androzoo-M12	39 767	125 225	31 489
Androzoo-M16	39 325	208 625	53 051

Table 2: Comparison of cryptographic API spread in benign vs. malicious datasets. The last column normalizes by the size of the datasets, allowing for direct comparison [18].

CryptoLint-Androzoo Comparison. The CryptoLint-B12 dataset resulted from scanning 145 095 samples for the presence of cryptographic API (and

Dataset	AES	DES	3DES	RC4	Blowfish	Unknown
CryptoLint-B12	**58.9%**	19.0%	8.8%	0.4%	1.9%	10.9%
BinSight-B16	**64.4%**	14.3%	1.1%	2.1%	0.9%	17.2%
Androzoo-B12	**52.4%**	16.9%	3.8%	0%	0.0%	26.8%
Androzoo-B16	**59.0%**	12.2%	2.0%	0.1%	0.0%	26.8%
Androzoo-M12	12.1%	**56.0%**	0.9%	0.0%	0.0%	31.0%
Androzoo-M16	**45.1%**	22.8%	2.1%	0.0%	0.0%	30.0%

Table 3: Distribution of symmetric ciphers in benign and malicious datasets with AES dominating all but Androzoo-M12 [18].

its misuse). The study concluded that 15 134 (10.4%) of APKs contain some cryptographic call sites. However, the subsequent BinSight study attributed 79.5% of these call sites to the ignored third-party packages, showing that the original CryptoLint study suffered from overcounting.

In contrast, we report that 27.4% of Androzoo-B12 contains cryptographic API call sites and nearly twice as much malware from Androzoo-M12 (53.1%). This highlights the extensive use of cryptographic API in malicious applications compared to the benign landscape. A closer examination of symmetric ciphers in Table 3 (also reported in [18])reveals considerable differences between malicious and benign datasets. AES dominates benign datasets with 58.9% in CryptoLint-B12 and 52.4% in Andozoo-B12. The situation is strikingly different in the malicious dataset. In Androzoo-M12, the most popular primitive is DES with 56% of call sites, followed by AES (12.1%) and 3DES (0.9%). We provide a more in-depth comparison of individual ciphers and their modes of operation in Appendix A.

BinSight-Androzoo Comparison. The BinSight paper aimed to answer what proportion of cryptographic API misuse can be attributed to third-party packages. The authors identified 638 distinct third-party packages in 115 683 unique samples in BinSight-B16, relying on the package name as an identifier. The authors attributed at least 90.7% of the call sites to third-party packages, underlying the need for their robust detection. Even after we discarded 9 870 third-party packages from Androzoo-M16, the malicious dataset still contains much more cryptographic API in the user-authored codebase. Again, the relations between Androzoo-M16 and BinSight-B16 are depicted in Table 2. Interestingly, in 2016, AES was dominant in Androzoo-M16 as well with 45.1% of call sites, followed by DES (22.8%). We depict the time evolution of AES vs. DES in Androzoo dataset in Figure 1 (also reported in [18]), showing that it was only in 2015 when AES outran DES in malicious applications.

6.5 Crypto API Categories in Malware

Apart from the comparison to benign applications, we also report a broad view of the distribution of cryptographic API in *malicious* applications, concentrating on the years 2012-2018, for which we can rely on a set of representative samples not clouded with high ratios of obfuscated applications.

Table 4 (also reported in [18]) illustrates that the majority of call sites from this period can be attributed to hash functions (66%) and symmetric encryption (26%), which leaves the rest of the categories rarely used. Nevertheless, we comment on our findings in all categories, observing the time evolution trends and showing the most prevailing primitives. We could not attribute 21% of the identified constructors to the exact cryptographic primitive (partial obfuscation) during our experiments. We still manage to pinpoint their presence and category, as the system-based API calls are challenging to obfuscate entirely.

Hash Functions. The hash functions are by far the most popular category of cryptographic API in malicious applications, as they are present in 40% of all studied APKs and responsible for 424 858 call sites in our dataset. Interestingly, the majority of the call sites resort to primitives MD5 or SHA-1 that were already shown to be broken [43,41]. Specifically, MD5 can be attributed to more than 80% of these call sites and does not lose any popularity in time. This may suggest that MD5 is either not meant to provide secure integrity protection for the authors or that the developers are unaware of its weakness. The time evolution of SHA-1 and SHA-256 points to the former case. Indeed, the overall dominant SHA-1 (almost 16% of call sites) is gradually decreasing over time in favor of the more secure SHA-256 (3% overall). In 2018, SHA-256 was present in more APKs (708) than SHA-1 (528). This phenomenon can mean that, when secure integrity protection is needed, more secure SHA-256 is nowadays being selected instead of SHA-1. Still, MD5 is preferred by malware creators for other use cases. Apart from the hash functions mentioned above, only SHA-512 and SHA-384 are represented in the dataset, but these are responsible for less than 1000 call sites in total.

Symmetric Encryption. A large portion of the symmetric encryption API landscape was already described in Section 6.4, but some important aspects were yet omitted. Overall, our dataset contains 165 994 symmetric encryption call sites distributed in approximately 20% of APKs. A large portion of the call sites (26%) is obfuscated. Besides AES and DES, only 3DES is used in more than 1000 APKs. We also report that the concept of password-based encryption is applied merely in 837 APKs. A closer look at the encryption modes offers an interesting perspective. Our observations confirm that the authors favor the default constructors (`"AES"` and `"DES"`) compared to constructors that specify encryption mode and padding (e.g., `"AES/CBC/PKCS5PADDING"`). The default constructors fall back into the ECB mode with PKCS#7 padding, which is (under most circumstances) considered insecure [30].

Public-key Encryption. The only asymmetric encryption scheme appearing in the Android API is RSA. The RSA encryption occurs in approximately 1.55% of all APKs in our dataset. Until 2013, RSA appeared very rarely, but then it peaked within two years at almost 1 800 APKs in 2015.

Digital Signature Algorithms. Surprisingly, digital signature algorithms occupy 4.5% of the malicious APKs and are present in 17 505 call sites. Considering possible applications of digital signature primitives in malware, this

Category	#call sites	%obfusc.	%APK
Hash functions	424 858	16.8%	39.7%
Symmetric enc.	165 994	25.9%	19.4%
Public-key enc.	13 262	25.9%	1.5%
Digital sig. alg.	17 505	81.4%	4.5%
MAC	11 661	46.4%	3.0%
PRNGs	10 381	6.6%	2.9%
Key agreement	87	29.9%	0%
Sum	646 018	21.5%	44.6%

Table 4: Popularity distribution of cryptographic API categories. The ratio of APKs for symmetric encryption and RSA is approximate since one cannot differentiate the obfuscated constructors of these two categories. Note that approximately 1% of APKs contain cryptographic API outside of these categories [18].

constitutes a rather large number. Despite the highest obfuscation rate among categories (81.43%), the `SHA1withRSA` primitive is responsible for almost 80% of the unobfuscated call sites. As in the case of the hash functions, `SHA256withRSA` is on the rise in time, first appearing in 2015 and steadily increasing the fraction of APKs it appears in ever since. Still, in 2018 it is less than four times probable to appear compared to `SHA1withRSA`.

While multiple schemes supporting elliptic curves over RSA or DSA are offered in the API, these are explicitly specified only in 19 APKs in total, with the first use appearing in 2014.

MAC Algorithms. The situation with MAC algorithms is similar to that of digital signature algorithms. The MAC systems are responsible for 11 661 call sites and are present in 3% of APKs. Still, a large portion (46.4%) of the call sites are obfuscated. Nevertheless, only two functions are called in more than 1% of the call sites – `HMACSHA1` and `HMACSHA256`. The former is heavily dominant throughout the studied timeline, being responsible for 70% of the MAC call sites.

PRNGs. The functionality of PRNGs is utilized in nearly 3% of the APKs, being responsible for 10 381 call sites. A relatively small fraction of the call sites (6.6%) are obfuscated, and virtually all unobfuscated call sites (over 90%) can be attributed to `SHA1PRNG`.

Key Agreement Protocols. The key agreement API's functionality consists purely of the Diffie-Hellman protocol (DH) for key exchange. Concerning the DH protocol parameters, we can only differentiate between the use of DH over finite fields or elliptic curves. The key agreement API appears only in 53 APKs over the nine years, occupying 87 call sites in total. 36 of the APKs use elliptic curves, whereas 20 APKs use obfuscated calls. Interestingly, only a single APK was detected to be explicitly using DH over finite fields. The dominant use of elliptic curves over finite fields is in contrast with the situation in digital signature algorithms.

Worth noting, we did not thoroughly explore how cryptographic primitives were employed in the *context* of the applications (e.g., to send SMS, encrypt data, et cetera). This analysis is extremely complex due to the variety of application contexts, and it is hardly feasible with static analysis. However, to give readers possible directions about the motivations for using cryptography in malware, we manually inspected a small subset of samples during our study. For the categories defined in Section 3.1 we documented the following use-cases: *(i) Hash functions* are generally used to fingerprint the attributes of a device (IMEI, Android version, etc.), to hash whole file or string, or to construct home-brew MACs or signature primitives; *(ii) Public-key encryption* was witnessed to provide hybrid encryption or to construct digital signature algorithms from its basic blocks; *(iii) Symmetric encryption* is used to encrypt files, as well as strings, and to obfuscate expressions directly in the source code. We also witnessed the use of *PRNG* to generate random keys (often with static seeds) or to provide nonces for more complex scenarios; *(iv)* Both *key agreement protocols* and *digital signature algorithms* were found to empower more complex network protocols, e.g., SASL (Simple Authentication and Security Layer [17]); *(v)* We report no surprising use-cases for *MAC* primitives that serve their original purpose of data authentication.

7 MACHINE LEARNING AND CRYPTOGRAPHIC API

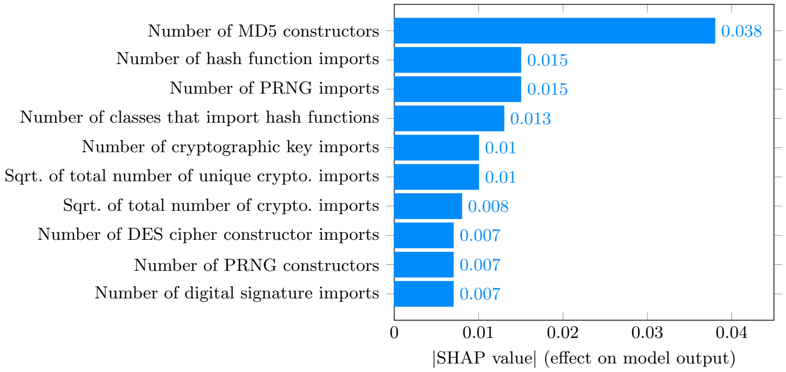


Fig. 2: A representation of 10 most influential features (their |SHAP| values are high, averaged over all samples). These represent the spots in cryptographic API with the largest difference in usage between malicious and benign samples. The x-axis shows the average effect on the model output in either direction. The model outputs values from 0 (benign) to 1 (malicious) [18].

In this section, we answer RQ.2: *How features related to cryptography affect the performances of Android malware detectors?* To do so, we analyze the outcome of the experiments outlined in Section 4.

Cryptography-Based Learning Model According to the preliminary results reported in Section 4, we trained a random forest model based only on cryptography-related features described in the same Section and compared its performance to `R-PackDroid`. To obtain a valid comparison, we replicated the experimental setup of the original `R-PackDroid` paper [26], considering10 thousand applications divided 50:50 into benign/malicious and split 50:50 into training/test sets. The proposed classifier achieved 62.4% F1 score on the malicious samples set (see also Table 5), showing that cryptographic information is discriminant enough to separate malicious from benign samples. Even though `R-PackDroid` performs significantly better than the proposed system[11], the proposed classifier was able to correctly identify 88/180 malicious samples that were misclassified as benign by `R-PackDroid` (with all 211 features). This shows that cryptographic API can assist the classification of samples that would otherwise fly under the radar of existing classifiers that does not include specific features related to cryptographic usage.

Explanations of Decisions Figure 2 (also reported in [18]) shows the 10 most influential features of the cryptographic-API classifier, based on the SHAP values calculated for the whole dataset (i.e., global explanations) according to the methodology described in Section 4. Model outputs are mapped as follows: values range from 0 (benign) to 1 (malicious), and the expected value of the model on a balanced dataset is hence 0.5. The SHAP value of a feature thus represents a deviation from this expected value after inspecting a particular feature.

It is rather interesting to see that the usage of certain hashing functions is discriminative w.r.t. maliciousness of the samples. More specifically, weak hash functions (MD5) are especially used in malicious samples (as also reported by the analysis in Section 6.4), and they constitute an important indicator of maliciousness. Additionally, the classifier is also sensitive to the general number of imported cryptographic functions. An increasing number of imported functions lead to an increasing suspicion of maliciousness. We can thus conclude that the statistical analysis reported in the previous Section is confirmed by the analysis of explanations provided by Android malware classifiers.

Concerning the local explanation, we present an example related to the malware samples with MD5 hash `e1001da40929df64443f6d4037aa3a9f`. VirusTotal classifies this sample as a riskware of type SMSpay. By extracting the local SHAP values (Figure 3, also reported in [18]), it is possible to see the significant importance of a DES encryption that steers the classifiers' decision towards maliciousness. Driven by this explanation, we manually disassembled the sample and looked for the usage of DES-related cryptographic API. We found that, in this case, DES is used to encrypt sensitive information, such as the phone device id, which is then subsequently exfiltrated to a remote server. This detail is especially useful to attract the attention of the analyst toward malicious operations carried

[11] Remember that our goal was not to build a better classifier but to show that it is possible to distinguish between malicious and benign Android applications by resorting to their cryptographic API usage only.

out by the sample. Also, note that this sample employs name obfuscation, and the required effort to carry out a similar analysis without such guidance would be higher. Explanations can thus provide further insights into the malicious behavior of malware samples, confirming once again that effective threat analysis requires the usage of multiple different tools.

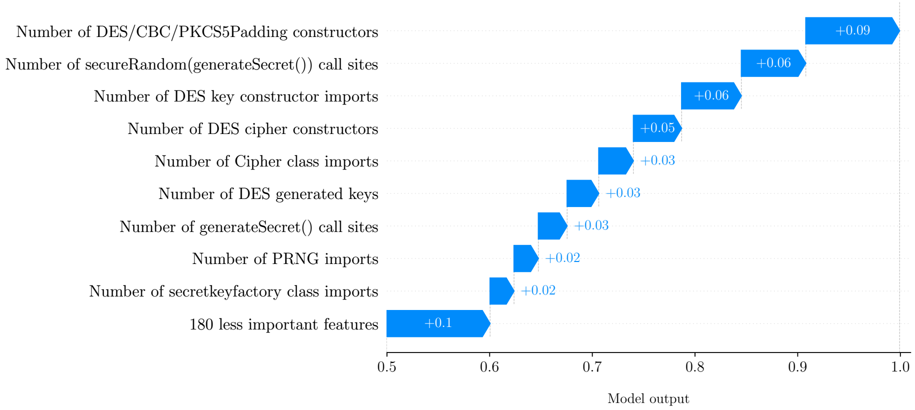


Fig. 3: Local impact of the 10 most influential features w.r.t the models' prediction *on a particular sample*. The shown APK is a malware sample of the SMSpay type. It exfiltrated DES-encrypted data through SMS. Apart from the features, the figure also depicts how the value of each feature shifts the models' output from a neutral score of 0.5 to the final output of 1 which labels the APK as malware [18].

Enhancing Existing Classifier with Cryptographic API Features In the original work, `R-PackDroid` achieved a 97% F1 score distinguishing between three classes (ransomware in addition to malware/benign). Leveraging on the source code provided by the authors, we achieved a 92.71% F1 score with `R-PackDroid` on the Androzoo dataset. Thus, enhancing this classifier would be a hard task, as it would result in a not significant performance gain. Hence, we decided to rely on a *light* version of `R-PackDroid` that employs a reduced feature set made up of 10 features, reducing the F1 score to 74.47% when averaged over different 10-tuples of features. We then added the cryptographic features following the methodology reported in Section 4.

This experimental setup allowed us to better appreciate the influence of crypto features in increasing the classifiers' performance. Reported results show that adding cryptographic features significantly improved both recall (+5.61%), and precision (+4.18%) of the classifier, which in turn increases the F1 score by +4.86%. See the summary in Table 5.

Classifier	# features	F1 score
Cryptographic API	189	62.40%
RPackDroid (full)	211	92.71%
RPackDroid (limited)	10	76.47%
RPackDroid + cryptoAPI	10 + 189	81.33%

Table 5: Comparison of the performance of malware classifiers without and with cryptographic API features. The performance metrics measure the result of the malicious samples. Enhancing the limited RPackDroid with cryptographic features causes a 4.18% increase in precision and 5.61% increase of recall on the Androzoo dataset, projecting into a 4.86% F1 score increase [18].

8 DISCUSSION AND LIMITATIONS

Our main goal was to provide a comprehensive overview of the role of cryptography in the analysis of Android applications and malware detection. The description of the experimental results reported in Section 6.4 clearly showed that recent malicious applications are characterized by a significant amount of obfuscation, thus preventing the extraction of detailed information about their usage of cryptography. We recognize that this is an inherent limitation of static analysis. While other static techniques such as program slicing may provide additional insights, we consider dynamic analysis as the only reliable way to cope with dynamic code loading and other types of packing. On the other hand, it is infeasible to analyze more than half a million of APKs with dynamic analysis only. Hence, we support our proposed approach as providing an effective balance between effectiveness, precision, and analysis time.

The results reported in this paper can be affected by possible biases that can be present in the data we analyzed. In particular, we did not have control over the contents of the Androzoo dataset. According to the indications provided by the Androzoo authors [1], we can safely rule out the presence of possible duplicates for applications coming from the same sources (e.g., the same stores). We point out that the risk of finding duplicates across stores is significantly lower for malicious applications than benign ones. Nevertheless, even if such duplicates were found, their number should not influence a large-scale analysis.

9 RELATED WORK

Most of the research on cryptographic API in Android is mainly focused on benign applications where the ultimate goal is to mitigate its misuse. Several steps are needed to achieve this, and the respective works usually treat one or two steps at a time. We can summarize these steps as follows: *(i)* Inferring the rules of cryptographic API misuse; *(ii)* Evaluation of cryptographic API misuse; *(ii)* Attribution of cryptographic API misuse; *(iv)* Automatic cryptographic API repairs. The following paragraphs discuss the related research for all these steps mentioned above.

Inferring Rules of Cryptographic API Misuse. In the area of inferring the rules of cryptographic API misuse, the goal is to create a list of specifications for developers and researchers that imply the insecure use of cryptography. Such rules can be crafted manually as done in [11,8,39]. However, this approach does not scale well, leading to the works [33,13] that attempt to infer these rules from git commits, conjecturing that newly introduced commits typically eliminate security vulnerabilities from the code. Surprisingly, Paletov et al. [33] reported success with this approach, whereas chronologically later work [13] commends against the initial assumption.

Evaluation of Cryptographic API Misuse. After having a set of rules that suggest security violations at hand, it is vital to explore these violations in the Android applications market. While more powerful dynamic analysis is employed in [8,39] to show that more than half of the examined applications violate the static set of rules, the application dataset is relatively small (size < 100). On the contrary, the static analysis approach used by Egele et al. in [11] allowed examining a large dataset of 145 thousand benign applications to reveal that 10.4% of them uses some form of cryptography. 88% of such applications were found to violate some rules of secure cryptography usage. These results were confirmed by a later study [31] that gathered a new dataset of 109 thousand APKs that contain at least one cryptographic API call and showed the analogical proportion of insecure applications. Static rules were substituted by a more sophisticated definition language in [19] where 10 000 Android applications were analyzed and misuses detected in over 95% of cases.

Some of the solutions above are impractical to run against large projects due to many false positives. This is treated by `CryptoGuard` [36] that prunes the alerts to achieve 98% precision and is successfully run against real-world projects. As of early 2022, an open-source project named `CRYLOGGER` [34] can well complement `CryptoGuard`, as it is based on dynamic analysis and was tested on a sufficiently large dataset (approx. 1800 applications). In 2021, the first systematic evaluation study [2] was published that allows measuring the quality of such detectors and reveals many flaws in their design or implementation.

Concentrating on the TLS protocol, this work from 2012 [12] analyzed 13 thousand Android applications to reveal inadequate TLS usage in 8% of cases. The authors also managed to launch 41 MiTM attacks against selected applications. Iterating on this effort, another paper [32] studied Network Security Configuration files[12] in Android. The authors revealed that 88% of applications employing custom settings downgrade the security compared to the default configuration. Also, the authors penetrated Google Play safeguards that are supposed to protect from publishing applications vulnerable to MiTM.

Attribution of Cryptographic API Misuse. Reliable third-party package detection is central for attribution of misuse. This problem has been addressed, e.g., in [42,24,4,4] where matching algorithms were proposed to reliably detect third-party libraries. As already discussed, a systematic review [44] then compared these detectors from various perspectives confirming that `LibRadar` is superior

[12] `developer.android.com/training/articles/security-config`

to others when used for large-scale analysis due to result quality comparable with the most precise tools, yet running much faster.

Automatic Cryptographic API Repairs. More distant to our research are papers that concentrated on automatic cryptographic API misuse repairs. From this area of research, we refer the reader to [23,45].

Study of Cryptography in Android Malware. We point out that all the aforementioned research results on the Android platform did not focus on the usage of cryptographic API in malicious applications or in its comparison to the benign landscape. The work presented in this paper aims to fill this knowledge gap.

10 CONCLUSIONS AND FUTURE WORK

The main motivation behind this research work is the qualitative observation of the increased use of cryptographic APIs in Android malware. Cryptography is used in various malware modules such as external communication, file encryption, etc. Moreover, cryptography is also employed to obfuscate the malware content and behavior.

We thus performed a quantitative evaluation based on collecting a large number of malware samples covering the past decade. We designed a system based on the static analysis of Android applications to assess the use of cryptographic APIs and computing the related statistical measures. The results of this first phase provided a clear picture of the evolution in the use of cryptographic APIs in Android malware and the difference between goodware and malware in the use of cryptographic APIs.

To get more quantitative information, we trained machine learning classifiers to discriminate between goodware and malware according to the statistical measures on cryptographic APIs computed in the first phase. Thus, we could assess the extent to which features related to the use of cryptographic APIs contribute to the discrimination between goodware and malware. Then, we also ranked the features according to the *explanation* of their influence in the final decision of the classifier.

The result of this analysis is twofold: on the one hand, the developed tool allows a deeper understanding of the internals of a malware sample. On the other hand, we showed that the highest-ranked features could be used to improve the classification performance of malware detectors. This is a clear advance with respect to the state of the art, as cryptographic features until now have been neglected in the design of Android malware detectors.

Reported results have been obtained through the analysis of 603 937 applications and the extraction of over 1 million call sites. The most prominent facts can be summarised as follows:

1. *Use of weak hash functions.* Most malicious applications featuring cryptographic routines resorted to weak MD5 hash functions.

2. *Late transition from DES to AES.* In the symmetric cipher category, malware authors switched from weak DES to modern AES only in 2015, while AES was the most popular cipher in benign samples already in 2012.
3. *Very limited use of third-party cryptographic libraries.* Android application authors favor using system-based libraries to deliver cryptographic functionality.
4. *Contrast between malicious and benign usage of cryptography.* Our study shows that cryptographic API is generally more frequent in malware than in benign samples (in relative measures).

The results in this paper open the door to several follow-up research projects. On the one hand, malware samples could be clustered into families according to their usage of cryptography. On the other hand, it would also be of interest to understand for which *main purpose* specific crypto-routines have been included, to better understand and profile the characteristics of malware authors.

Finally, as the results of this work are entirely based on static analysis, it could be complemented by dynamic analysis to check if our findings also hold for packed and obfuscated applications.

ACKNOWLEDGEMENTS

Davide Maiorca was supported by the project PON AIM Research and Innovation 2014–2020 - Attraction and International Mobility, funded by the Italian Ministry of Education, University and Research; and by the European cybersecurity pilot CyberSec4Europe. Giorgio Giacinto was supported by Fondazione di Sardegna under the project “TrustML: Towards Machine Learning that Humans Can Trust”, CUP: F73C22001320007. Vashek Matyas was supported by Czech Science Foundation project GA20-03426S. Adam Janovsky was supported by Invasys company. We are grateful to Jonas Konecny who ran the initial machine-learning experiments. We also thank Avast for providing the dynamic-analysis tool `apklab.io`.

References

1. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: AndroZoo: Collecting millions of Android apps for the research community. In: Proc. of MSR ’16. pp. 468–471. ACM (2016)
2. Ami, A.S., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., Nadkarni, A.: Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. arXiv:2107.07065 [cs] (Aug 2021)
3. Anthony, D., Geoffroy, G.: Androguard (2012), `https://github.com/androguard/androguard`, accessed on August 4, 2019
4. Backes, M., Bugiel, S., Derr, E.: Reliable third-party library detection in android and its security applications. In: Proc. of CCS ’16. pp. 356–367. ACM (2016)
5. Bauer, V.: Android Arsenal (2014), `https://android-arsenal.com`, June 5, 2020

6. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. Journal of CRYPTOLOGY **4**(1), 3–72 (1991)
7. BusinessOfApps: Android statistics. `http://businessofapps.com/data/android-statistics` (2022), `https://www.businessofapps.com/data/android-statistics/`
8. Chatzikonstantinou, A., Ntantogian, C., Karopoulos, G., Xenakis, C.: Evaluation of Cryptography Usage in Android Applications. In: Proc. of EAI BCT '16. pp. 83–90. ACM (2016)
9. Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H.: Stormdroid: A streaminglized machine learning-based system for detecting android malware. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 377–388. ASIA CCS '16, ACM, New York, NY, USA (2016)
10. Daniel, A., Michael, S., Malte, H., Hugo, G., Rieck, K.: Drebin: Efficient and explainable detection of android malware in your pocket. In: Proceedings 2014 Network and Distributed System Security Symposium. pp. 23–26. The Internet Society, San Diego, CA (2014)
11. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proc. of CCS'13. pp. 73–84. ACM (2013)
12. Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., Freisleben, B.: Why eve and mallory love android: An analysis of android SSL (in)security. In: Proc. of CCS '12. pp. 50–61. ACM (2012)
13. Gao, J., Kong, P., Li, L., Bissyande, T.F., Klein, J.: Negative Results on Mining Crypto-API Usage Rules in Android Apps. In: Proc. of MSR '19. pp. 388–398. IEEE (2019)
14. Google: Android Cryptography API Guide (2020), `https://developer.android.com/guide/topics/security/cryptography`, accessed on March 4, 2020
15. Google, i.: Conscrypt - a java security provider (2013), `https://github.com/google/conscrypt`, accessed on June 5, 2020
16. Hoffmann, J., Rytilahti, T., Maiorca, D., Winandy, M., Giacinto, G., Holz, T.: Evaluating analysis tools for android apps: Status quo and robustness against obfuscation. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. pp. 139–141. Association for Computing Machinery, New York, NY, USA (2016)
17. Isode Limited, OpenLDAP Foundation: RFC 4422 - simple authentication and security layer (sasl). `http://tools.ietf.org/html/rfc4422` (2006), March 2, 2022
18. Janovsky., A., Maiorca., D., Macko., D., Matyas., V., Giacinto., G.: A longitudinal study of cryptographic api: A decade of android malware. In: Proceedings of the 19th International Conference on Security and Cryptography - SECRYPT,. pp. 121–133. INSTICC, SciTePress (2022). https://doi.org/10.5220/0011265300003283
19. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In: Proc. of ECOOP '18. pp. 10:1–10:27. LIPIcs vol. 109, LZI (2018)
20. Kursa, M.B., Rudnicki, W.R., et al.: Feature selection with the boruta package. J Stat Softw **36**(11), 1–13 (2010)
21. Legion of the Bouncy Castle Inc.: The Legion of the Bouncy Castle. `https://www.bouncycastle.org/java.html` (2020), accessed on April 6, 2020
22. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. In: Proc. of NIPS '17, pp. 4765–4774. Curran Associates, Inc. (2017), `http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf`

23. Ma, S., Lo, D., Li, T., Deng, R.H.: CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In: Proc. of ASIACCS '16. pp. 711–722. ACM, Xi'an, China (2016)
24. Ma, Z., Wang, H., Guo, Y., Chen, X.: LibRadar: Fast and accurate detection of third-party libraries in Android apps. In: Proc. of ICSE '16. pp. 653–656. ACM, Austin, Texas (2016)
25. Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth attacks: An extended insight into the obfuscation effects on android malware. Computers & Security **51**(C), 16–31 (Jun 2015)
26. Maiorca, D., Mercaldo, F., Giacinto, G., Visaggio, C.A., Martinelli, F.: R-PackDroid: API package-based characterization and detection of mobile ransomware. In: Proc. of SAC '17. pp. 1718–1723. ACM (2017)
27. McAfee Labs: McAfee labs threats report, august 2019. `http://mcafee.com/enterprise/en-us/threat-center/mcafee-labs/reports.html` (2019), `http://mcafee.com/enterprise/en-us/threat-center/mcafee-labs/reports.html`, March 7, 2022
28. Melis, M., Maiorca, D., Biggio, B., Giacinto, G., Roli, F.: Explaining black-box android malware detection. In: 26th European Signal Processing Conference, EUSIPCO 2018. pp. 524–528. IEEE, Rome, Italy (2018)
29. Melis, M., Scalas, M., Demontis, A., Maiorca, D., Biggio, B., Giacinto, G., Roli, F.: Do gradient-based explanations tell anything about adversarial robustness to android malware? Int. J. Mach. Learn. Cybern. **13**(1), 217–232 (2022). https://doi.org/10.1007/s13042-021-01393-7, `https://doi.org/10.1007/s13042-021-01393-7`
30. Menezes, A.J., Katz, J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied cryptography. CRC press (1996)
31. Muslukhov, I., Boshmaf, Y., Beznosov, K.: Source Attribution of Cryptographic API Misuse in Android Applications. In: Proc. of ASIACCS '18. pp. 133–146. ACM (2018)
32. Oltrogge, M., Huaman, N., Amft, S., Acar, Y., Backes, M., Fahl, S.: Why eve and mallory still love android: Revisiting TLS (In)Security in android applications. In: Proc. of USENIX '21. pp. 4347–4364. USENIX (2021)
33. Paletov, R., Tsankov, P., Raychev, V., Vechev, M.: Inferring crypto API rules from code changes. In: Proc. of PLDI '18. pp. 450–464. ACM (2018)
34. Piccolboni, L., Guglielmo, G.D., Carloni, L.P., Sethumadhavan, S.: CRYLOGGER: Detecting Crypto Misuses Dynamically. In: Proc. of IEEE SP '21. pp. 1972–1989. IEEE (2021)
35. Platform, J.: Java Cryptography Architecture (JCA) Reference Guide (2017), `https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html`, accessed on March 4, 2020
36. Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.D.: CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In: Proc. of CCS '19. pp. 2455–2472. ACM (2019)
37. Salem, A.: Towards accurate labeling of Android apps for reliable malware detection. arXiv preprint arXiv:2007.00464 (2020)
38. Shapley, L.: A value for n-person games. contributions to the theory of games. Annals of mathematics studies (2) (1953)
39. Shuai, S., Guowei, D., Tao, G., Tianchang, Y., Chenjie, S.: Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In: Proc. of DASC '14. pp. 75–80. IEEE (2014)

40. skylot: Jadx decompiler (2020), `https://github.com/skylot/jadx`, Dec. 15, 2019
41. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full SHA-1. In: Proc. of CRYPTO '17. pp. 570–596. Springer (2017)
42. Wang, H., Guo, Y., Ma, Z., Chen, X.: WuKong: A scalable and accurate two-phase approach to Android app clone detection. In: Proc. of ISSTA '15. pp. 71–82. ACM (2015)
43. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Proc. of EUROCRYPT '05. pp. 19–35. Springer (2005)
44. Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., Xu, Y., Luo, X., Liu, Y.: Automated third-party library detection for Android applications: are we there yet? In: Proc. of ASE '20. pp. 919–930. ACM (Dec 2020)
45. Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., Gu, D.: Embroidery: Patching Vulnerable Binary Code of Fragmentized Android Devices. In: Proc. of ICSME '17. pp. 47–57. IEEE (2017)

Symmetric encryption scheme	Androzoo-M12	CryptoLint-B12
DES*	6356	741
DES/CBC/PKCS5Padding	1203	205
AES*	924	4803
AES/CBC/PKCS5Padding	786	5878
DESede/ECB/PKCS5Padding	231	473
AES/ECB/PKCS5Padding	122	443
DESede*	107	501
DES/ECB/PKCS5Padding	93	221
DES/ECB/NoPadding	68	1151
AES/CBC/NoPadding	43	468
AES/ECB/NoPadding	41	220
AES/CBC/PKCS7Padding	37	235
AES/CFB8/NoPadding	24	104
AES/ECB/PKCS7Padding	1	155
Sum AES where freq. > 100	1832	12306
Sum DES where freq. > 100	7559	2318
Sum DESede where freq. > 100	338	974
Sum where freq. > 100	9729	15598

Table 6: Comparison of distribution of symmetric encryption schemes in malicious vs. benign applications (Androzoo-M12 and CryptoLint-B12). The frequency of malicious encryption schemes was normalized to fit the size of the benign dataset. In the benign set, only the schemes with frequency > 100 were taken. There is no prevalent malicious scheme (freq. > 100) that would not appear in the benign dataset. The default schemes marked with * symbol fall back into the ECB mode with PKCS7 padding.

A Detailed comparison of symmetric ciphers between Androzoo-M12 and CryptoLint-B12

Table 6 displays an in-depth comparison between symmetric encryption schemes in the datasets CryptoLint-B12 and Androzoo-M12. It should be stressed that even though the absolute number of call sites in CryptoLint-B12 is higher (15 598) than in Androzoo-M12 (9729), this comparison is severely skewed by the overall distribution characteristics of CryptoLint-B12 vs. Androzoo-M12. In other words, it takes 145 thousand of benign applications (where only each fifth call originates from user-defined codebase) to get 15 thousand calls, whereas 34 thousand of malicious applications would provide a similar number of symmetric encryption API call sites.