# A Longitudinal Study of Cryptographic API: A Decade of Android Malware

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

[1]*Masaryk University, Czech Republic*
[2]*University of Cagliari, Italy*

Keywords: Cryptographic API, Malware, Android, Malware Detection.

Abstract: Cryptography has been extensively used in Android applications to guarantee secure communications, conceal critical data from reverse engineering, or ensure mobile users' privacy. Various system-based and third-party libraries for Android provide cryptographic functionalities, and previous works mainly explored the misuse of cryptographic API in benign applications. However, the role of cryptographic API has not yet been explored in Android malware. This paper performs a comprehensive, longitudinal analysis of cryptographic API in Android malware. In particular, we analyzed 603 937 Android applications (half of them malicious, half benign) released between 2012 and 2020, gathering more than 1 million cryptographic API expressions. Our results reveal intriguing trends and insights on how and why cryptography is employed in Android malware. For instance, we point out the widespread use of weak hash functions and the late transition from insecure DES to AES. Additionally, we show that cryptography-related characteristics can help to improve the performance of learning-based systems in detecting malicious applications.

## 1 INTRODUCTION

The Android operating system has spread worldwide during the last decade, reaching almost 3 billion users in 2021 (BusinessOfApps, 2022). At the same time, security threats against Android have multiplied, as recent reports showed (McAfee Labs, 2019).

Most Android applications employ cryptographic primitives 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 that malware authors may leverage cryptography in a plethora of 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 related to the misuse of cryptographic application programming interface (API) in benign Android applications, i.e., on finding and eliminating vulnerabilities in the employed crypto-routines that may allow attackers to obtain sensitive information (Egele et al., 2013; Muslukhov et al., 2018; Chatzikonstantinou et al., 2016; Shuai et al., 2014).

To the best of our knowledge, however, no study explored how cryptography is currently employed in *malicious* applications. In this paper, we answer two important 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:** Can information about cryptography improve Android malware detection?

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. 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[1], scalable approach that inspects (among others) the usage of hash functions, symmetric

---

[1]The code is acessible from github.com/adamjanovsky/AndroidMalwareCrypto.

121

and public-key encryption, PRNGs, etc. In total, we inspect over $10^6$ of cryptographic API expressions.

Second, we show that cryptographic features can be used to augment the performances of state-of-the-art malware detectors, demonstrating their discriminant power in distinguishing malicious and benign applications. We also employ techniques inherited from the interpretation of learning models to point out possible connections between cryptographic API and malicious actions.

The attained results show many intriguing and surprising trends. For example, in contrast to benign applications, malware authors do not typically resort to strong cryptography to perform their actions. We show that malware often favors the use of cryptographically defeated primitives, e.g., weak hash functions MD5 (Wang and Yu, 2005) or SHA-1 (Stevens et al., 2017), or symmetric encryption scheme DES (Biham and Shamir, 1991). 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 that 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: In Section 2, we describe the methodology of our analysis. In Section 3, we answer the first research question. Section 4 discusses cryptographic API in relation to malware detection. The limitations of our study are discussed in Section 5. Section 6 describes the related work and Section 7 closes the paper with the conclusions.

We also deliver an extended technical report for this paper on arXiv[2].

## 2 METHODOLOGY

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

### 2.1 Problem Formalization

We organize the problem formalization in two parts: part one treats the definition of the crypto-routines of

_____
[2]https://arxiv.org/abs/2205.05573.

interest for our analysis, 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). Notably, in this analysis, we discard custom cryptographic functions that users or third parties may implement. Automatic detection of such functions would be a complex task in the context of 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 aims 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 perform two steps: *(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 (Wang et al., 2015) that more than 60% of Android APK[3] 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 (Wang et al., 2015; Ma et al., 2016b; Backes et al., 2016). It does not suffice to merely search for the `import` clauses in the decompiled source code since the non-system packages could be renamed. This scenario is especially frequent in malicious applications, as the authors aim to defend against any forensics. Inspired by the systematic review of third-party package detectors (Zhan et al., 2020), we opted to tackle this with `LibRadar`, a popular third-party libraries detection tool that utilizes clustering and complex signatures to recognize such libraries (Ma et al., 2016b). In this review, `LibRadar` achieves the highest precision and second-highest recall while it takes 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, not relying on package names. Consequently, `LibRadar` can identify obfuscated packages. Using `LibRadar`[4], we filter the identified third-party packages of an APK from subsequent cryptographic API analysis.

## 2.2 Crypto API Extraction Pipeline

Our system generates a comprehensive report of the embedded cryptographic API, given an application dataset. As an input, configuration files for the to-be-conducted experiment are taken. 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

---

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

[4]Since `LibRadar` requires large Redis database to run (preventing parallelization), we actually leveraged its lightweight version `LiteRadar`. Prior to 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.

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 so that one can alter its inner workings to extract further valuable insights from the APKs.

## 2.3 Dataset

To gain an all-around view of the cryptographic API landscape in Android applications, we leverage the Androzoo dataset (Allix et al., 2016). Currently, Androzoo is the 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, 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[5], which should reliably eliminate benign files deemed malicious, as reported by Salem (Salem, 2020). 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 (Allix et al., 2016) to avoid over-counting.

## 2.4 Cryptography and Machine Learning

Statistics about cryptographic usage are undoubtedly helpful in pointing out differences between benign and malicious applications. Another intriguing question to

---

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

explore is *whether such statistics can be useful to recognize malicious samples from benign ones effectively.* To answer this question, we propose three methods that employ machine learning techniques, described in the following.

### 2.4.1 Cryptographic Learning Model

The first technique consists of defining a learning-based system whose structure is inspired by other popular detection systems (Daniel et al., 2014; Chen et al., 2016; Maiorca et al., 2017). In particular, the proposed 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 2.2; *(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 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 a feature selection algorithm. The dataset with candidate features is split in a 9:1 ratio into training/test sets. Then, we apply two feature selection methods to drop uninformative features. 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 that are deemed uninformative by Boruta (Kursa et al., 2010). 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. The feature selection process yields 189 features that are used for learning.

To choose the best family of models for discriminating between malicious and benign samples on our dataset, we experimented with naive Bayes, logistic regression, support vector machines with linear kernel, random forest, gradient boosted decision trees, and multilayer perceptron. We tuned the classifiers' hyperparameters using 10-fold cross-validation on the

training dataset, optimizing for the F1 score. Subsequent evaluation yielded a random forest (which works as a majority-voting ensemble of decision trees trained on different subsets of the data) as the best-performing classifier (w.r.t. F1 score). We do not report the entire analysis here for brevity, and we stick to the random forest in the rest of the paper.

### 2.4.2 Explaining the Learning Model

To further advance the understanding of cryptographic API in Android detection, we explain the predictions of the just described cryptographic classifier. To do so, we follow the techniques of explainable machine learning, a field that attempts to understand why a classifier makes certain decisions. We study both *global* and *local* importances of the features extracted by the classifier, in a similar fashion to what has been proposed in previous work (Melis et al., 2018; Melis et al., 2022). A global analysis evaluates the impact of the features averaged over the whole dataset, while local analysis evaluates the impact of the features on specific samples.

To interpret the classifier's predictions, we used Shapley additive explanations (SHAP) (Lundberg and Lee, 2017) that are successfully used outside of computer science. 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 (Shapley, 1953) 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, dummy, and additivity.

### 2.4.3 Classifier Enhancing

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` (Maiorca et al., 2017), 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 fea-

tures. 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`[6]; *(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 4.

# 3 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 2.2. 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.

## 3.1 General API Distribution

### 3.1.1 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 1. 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 increasing ratio of obfuscated malicious applications. To confirm this hypothesis, we randomly sampled 4444 malicious applications from 2018-2020 that allegedly contain no cryptographic API or third-party libraries. We dissected them using commercial dynamic analysis tool `apklab`[7] and searched for clues of obfuscation. We

identified that 98% applications use some form of Android packer, with `jiagu` being most popular. Each packed application also uses reflection API and dynamic library loading, tools that 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 5.

### 3.1.2 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 1. 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 1). 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.

Apart from Android API, cryptographic functions can also be delivered by third-party libraries, typically adopted to integrate functionality missing in system-based libraries. With expert knowledge, we identified 13 Java and 10 native third-party cryptographic libraries for which we searched in the samples (see paper artifacts for the full list). 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).

---

[6]The tuples were sampled in advance to avoid repetition.
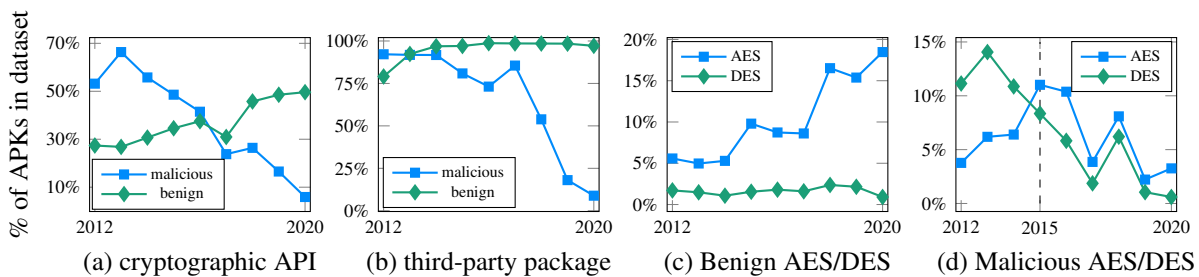[7]Kindly provided by Avast, available at http://apklab.io.

Figure 1: Time evolution of important dataset characteristics. The y-axis show the percentage of APKs exhibiting 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 outran by AES.

### 3.1.3 Crypto API in Goodware and Malware

We now describe the general prevalence of cryptographic API in the dataset presented in Section 2.3 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 (Egele et al., 2013) that we refer to as CryptoLint-B12; *(ii)* A dataset collected in 2016 as a part of Binsight study (Muslukhov et al., 2018) 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 2, our goal is to analyze only cryptographic APIs contained in user-defined code. From this respect, both (Egele et al., 2013; Muslukhov et al., 2018) 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 1. It can be seen that the malicious datasets have a dramatically higher density of cryptographic API call sites than their benign counterparts.

**CryptoLint-Anndrozoo Comparison.** The CryptoLint-B12 dataset resulted from scanning 145 095 samples for the presence of cryptographic API (and 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%), too. This highlights the extensive use of cryptographic API in malicious applications compared to the benign landscape. A closer examination of symmetric ciphers in Table 2 also 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%).

**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 1. 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, showing that it was only in 2015 when AES outran DES in malicious applications.

## 3.2 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 dispose of

Table 1: Comparison of cryptographic API spread in benign vs. malicious datasets. The last column normalizes by the size of the datasets, allowing for direct comparison.

| 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: Distribution of symmetric ciphers in benign and malicious datasets with AES dominating all but Androzoo-M12.

| 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% |

representative samples not clouded with high ratio of obfuscated applications.

Table 3 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 even their 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 (Wang and Yu, 2005; Stevens et al., 2017). Specifically, MD5 can be attributed to more than 80% of these call sites and does not lose any of its 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 the 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 3.1.3, 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 (Menezes et al., 1996).

**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 to 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 constitutes a rather large number. De-

Table 3: 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.

| 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% |

spite 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 signatures 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 2.1 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 (Isode Limited and OpenLDAP Foundation, 2006)); *(v)* We report no surprising use-cases for *MAC* primitives that serve their original purpose of data authentication.

# 4 MACHINE LEARNING AND CRYPTOGRAPHIC API

In this section, we answer RQ.2: *Can information about cryptography improve Android malware detection?* To do so, we analyze the outcome of the experiments outlined in Section 2.4.

### 4.0.1 Cryptography-based Learning Model

We trained a random forest model based only on cryptography-related features described in Section 2.4 and compared its performance to `R-PackDroid`. To obtain a valid comparison, we replicated the experimental setup of the original `R-PackDroid` paper (Maiorca et al., 2017), taking 10 thousand applications divided 50:50 into benign/malicious, and split 50:50 into training/test set. Our classifier achieved 62.4% F1 score on the malicious samples (see also Table 4), showing that cryptographic information is discriminant enough to separate malicious from benign samples. Even though that `R-PackDroid` performs sig-
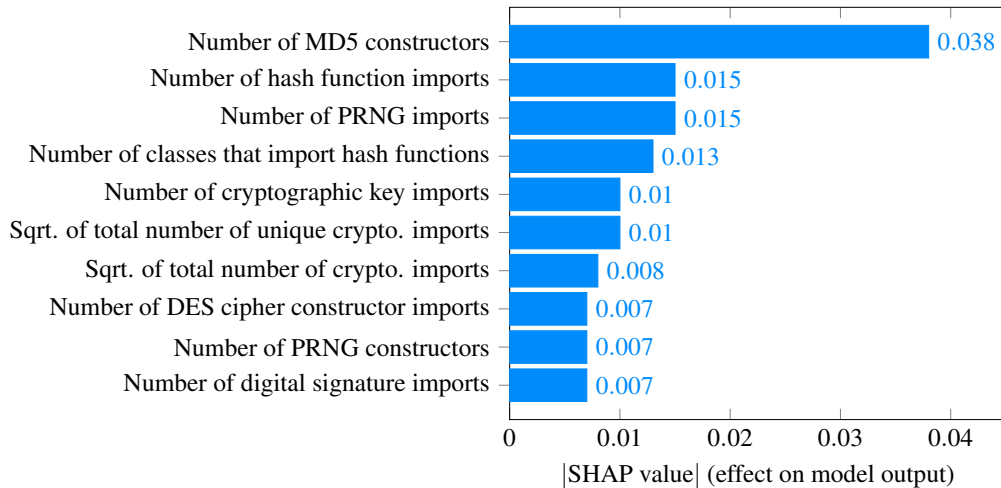
Figure 2: A representation of 10 most influential features (their |SHAP| values are high, averaged over all samples). These represent the spots in cryptographic API with the largest difference in usage between malicious and benign samples. The x-axis shows the average effect on the model output in either direction. The model outputs values from 0 (benign) to 1 (malicious).

nificantly better than our system[8], our 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.

### 4.0.2 Explanations of Decisions

In Figure 2, we present the 10 most influential features of the cryptographic-API classifier, based on the SHAP values calculated for the whole dataset (global explanation) according to the methodology from Section 2.4. Note that the model outputs values 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 3.1), and they constitute an important indicator of maliciousness. Additionally, the classifier observes the general number of imported cryptographic functions. An increasing number of imported functions lead to an increasing suspicion of maliciousness.

Concerning the local explanation, we present an example related to the malware with MD5 hash

---

Table 4: Comparison of 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 4.18% increase of precision and 5.61% increase of recall on the Androzoo dataset, projecting into 4.86% F1 score increase.

| 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% |

`e1001da40929df64443f6d4037aa3a9f`. VirusTotal classifies this sample as a riskware of type SMSpay. By extracting the local SHAP values (Figure 3), it is possible to see 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 towards malicious operations carried out by the sample. Also, note that this sample employs name obfuscation, and the required effort to carry out similar analysis without such guidance would be higher.

### 4.0.3 Enhancing Existing Classifier with Cryptographic API Features

In the original work, `R-PackDroid` achieved 97% F1 score distinguishing between three classes (ran-

---

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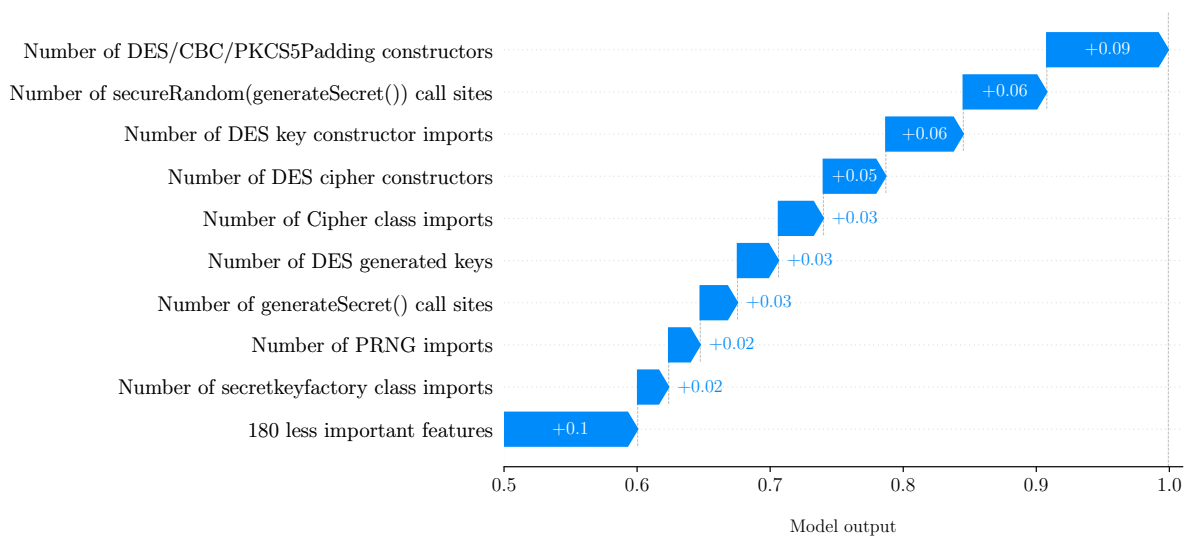
Figure 3: Local impact of 10 most influential features w.r.t the models' prediction *on a particular sample*. The shown APK is a malware of SMSpay type. It exfiltrated DES-encrypted data through SMS. Apart from the features, the figure also depicts how the value of each feature shifts the models' output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

somware in addition to malware/benign). Leveraging the source code of authors, we achieved 92.71% F1 score with `R-PackDroid` on the Androzoo dataset. Thus, enhancing this classifier would not bring a big performance gain. Hence, we decreased the number of features employed by the classifier to 10, reducing F1 score to 74.47% when averaged over different 10-tuples of features. We then added the cryptographic features following the methodology from Section 2.4.

We document an increase of classifiers' performance when enhanced with crypto features. Our results demonstrate that, while the margin for improvement is not huge, 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 4.

## 5 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. As explained in Section 3.1, we found a significant amount of obfuscation in recent malicious applications, which prevented us from providing 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 believe that dynamic analysis is the only reliable way to defeat dy-

namic code loading and other types of packing. However, it is infeasible to analyze more than half million of APKs with dynamic analysis. Hence, we believe that our proposed approach constitutes the right balance between effectiveness, precision, and analysis time.

We also point out that possible biases 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 (Allix et al., 2016), 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.

## 6 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 (Egele et al., 2013; Chatzikonstantinou et al., 2016; Shuai et al., 2014). However, this approach does not scale well, leading to the works (Paletov et al., 2018; Gao et al., 2019) 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. (Paletov et al., 2018) reported success with this approach, whereas chronologically later work (Gao et al., 2019) 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 (Chatzikonstantinou et al., 2016; Shuai et al., 2014) 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 (Egele et al., 2013) 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 rule of secure cryptography usage. These results were confirmed by a later study (Muslukhov et al., 2018) 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 (Krüger et al., 2018) 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` (Rahaman et al., 2019) that prunes the alerts to achieve 98% precision and is successfully run against real-world projects. As of early 2022, an open-source `CRYLOGGER` (Piccolboni et al., 2021) 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 (Ami et al., 2021) was published that allows to measure the quality of such detectors and reveals many flaws in their design or implementation.

Concentrating on the TLS protocol, the work (Fahl et al., 2012) from 2012 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, an article (Oltrogge et al., 2021) studied Network Security Configuration files[9] 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 (Wang et al., 2015; Ma et al., 2016b; Backes et al., 2016; Backes et al., 2016) where matching algorithms were proposed to reliably detect third-party libraries. As already discussed, a systematic review (Zhan et al., 2020) then compared these detectors from various perspectives confirming that `LibRadar` is superior 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 (Ma et al., 2016a; Zhang et al., 2017).

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

# 7 CONCLUSIONS AND FUTURE WORK

In this paper, we provide a longitudinal analysis of how cryptographic API was typically employed in Android malware in the years 2012-2020. We analyzed 603 937 applications and extracted over 1 million call sites that have been extensively analyzed by extracting various statistics. The attained results showed the following:

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.* It is worth pointing out one aspect of comparison with benign samples. In the symmetric cipher category, malware authors adopted weak DES to modern AES only

---

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

in 2015, while AES was the most popular cipher in benign samples already in 2012.

3. *Very limited use of third-party cryptographic libraries.* Our analysis showed that 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).

Building a cryptography-based machine learning model, we showed a significant difference in cryptography deployment between benign and malware samples. We demonstrated that a learning-based model based on cryptography alone could separate between benign and malicious samples with good performance. Moreover, we showed that explaining the decision of the classifier can expose intriguing links between cryptography and malicious actions that are typically carried out by malware. In particular, these techniques constitute a valid resource to guide the analysts towards discovering critical characteristics of the examined malicious samples. Moreover, cryptographic features until now neglected can be employed to further improve state-of-the-art malware detectors.

Our results open door to various follow-up work. For instance, it would be interesting to cluster malware samples into families based on its usage of cryptography. Likewise, future work can more closely focus on understanding *for what purpose* specific crypto-routines are employed in Android malware, thus better understanding and profiling the characteristics of malware authors. This could, for instance, confirm or dismiss our conjecture that weak hash functions are not used for integrity-critical operations in malware. Last but not least, one could employ more powerful dynamic analysis to check if our findings hold also for packed and obfuscated applications.

## ACKNOWLEDGEMENTS

## REFERENCES

Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). AndroZoo: Collecting millions of Android apps for the research community. In *Proc. of MSR '16*, pages 468–471. ACM.

Ami, A. S., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., and Nadkarni, A. (2021). Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. *arXiv:2107.07065 [cs]*.

Backes, M., Bugiel, S., and Derr, E. (2016). Reliable third-party library detection in android and its security applications. In *Proc. of CCS '16*, pages 356–367. ACM.

Biham, E. and Shamir, A. (1991). Differential cryptanalysis of DES-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72.

BusinessOfApps (2022). Android statistics. http://businessofapps.com/data/android-statistics.

Chatzikonstantinou, A., Ntantogian, C., Karopoulos, G., and Xenakis, C. (2016). Evaluation of Cryptography Usage in Android Applications. In *Proc. of EAI BCT '16*, pages 83–90. ACM.

Chen, S., Xue, M., Tang, Z., Xu, L., and Zhu, H. (2016). 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*, ASIA CCS '16, pages 377–388, New York, NY, USA. ACM.

Daniel, A., Michael, S., Malte, H., Hugo, G., and Rieck, K. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings 2014 Network and Distributed System Security Symposium*, pages 23–26, San Diego, CA. The Internet Society.

Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013). An empirical study of cryptographic misuse in android applications. In *Proc. of CCS'13*, pages 73–84. ACM.

Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben, B. (2012). Why eve and mallory love android: An analysis of android SSL (in)security. In *Proc. of CCS '12*, pages 50–61. ACM.

Gao, J., Kong, P., Li, L., Bissyande, T. F., and Klein, J. (2019). Negative Results on Mining Crypto-API Usage Rules in Android Apps. In *Proc. of MSR '19*, pages 388–398. IEEE.

Isode Limited and OpenLDAP Foundation (2006). RFC 4422 - simple authentication and security layer (sasl). http://tools.ietf.org/html/rfc4422. March 2, 2022.

Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. (2018). CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *Proc. of ECOOP '18*, LIPIcs vol. 109, pages 10:1–10:27. LZI.

Kursa, M. B., Rudnicki, W. R., et al. (2010). Feature selection with the boruta package. *J Stat Softw*, 36(11):1–13.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *Proc. of NIPS '17*, pages 4765–4774. Curran Associates, Inc.

Ma, S., Lo, D., Li, T., and Deng, R. H. (2016a). CDRep: Automatic Repair of Cryptographic Misuses in An-

droid Applications. In *Proc. of ASIACCS '16*, pages 711–722, Xi'an, China. ACM.

Ma, Z., Wang, H., Guo, Y., and Chen, X. (2016b). LibRadar: Fast and accurate detection of third-party libraries in Android apps. In *Proc. of ICSE '16*, pages 653–656, Austin, Texas. ACM.

Maiorca, D., Mercaldo, F., Giacinto, G., Visaggio, C. A., and Martinelli, F. (2017). R-PackDroid: API package-based characterization and detection of mobile ransomware. In *Proc. of SAC '17*, pages 1718–1723. ACM.

McAfee Labs (2019). McAfee labs threats report, august 2019. http://mcafee.com/enterprise/en-us/threat-center/mcafee-labs/reports.html. March 7, 2022.

Melis, M., Maiorca, D., Biggio, B., Giacinto, G., and Roli, F. (2018). Explaining black-box android malware detection. In *26th European Signal Processing Conference, EUSIPCO 2018*, pages 524–528, Rome, Italy. IEEE.

Melis, M., Scalas, M., Demontis, A., Maiorca, D., Biggio, B., Giacinto, G., and Roli, F. (2022). Do gradient-based explanations tell anything about adversarial robustness to android malware? *Int. J. Mach. Learn. Cybern.*, 13(1):217–232.

Menezes, A. J., Katz, J., Van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of Applied cryptography*. CRC press.

Muslukhov, I., Boshmaf, Y., and Beznosov, K. (2018). Source Attribution of Cryptographic API Misuse in Android Applications. In *Proc. of ASIACCS '18*, pages 133–146. ACM.

Oltrogge, M., Huaman, N., Amft, S., Acar, Y., Backes, M., and Fahl, S. (2021). Why eve and mallory still love android: Revisiting TLS (In)Security in android applications. In *Proc. of USENIX '21*, pages 4347–4364. USENIX.

Paletov, R., Tsankov, P., Raychev, V., and Vechev, M. (2018). Inferring crypto API rules from code changes. In *Proc. of PLDI '18*, pages 450–464. ACM.

Piccolboni, L., Guglielmo, G. D., Carloni, L. P., and Sethumadhavan, S. (2021). CRYLOGGER: Detecting Crypto Misuses Dynamically. In *Proc. of IEEE SP '21*, pages 1972–1989. IEEE.

Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., and Yao, D. D. (2019). CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *Proc. of CCS '19*, pages 2455–2472. ACM.

Salem, A. (2020). Towards accurate labeling of Android apps for reliable malware detection. *arXiv preprint arXiv:2007.00464*.

Shapley, L. (1953). A value for n-person games. contributions to the theory of games. Annals of mathematics studies (2).

Shuai, S., Guowei, D., Tao, G., Tianchang, Y., and Chenjie, S. (2014). Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In *Proc. of DASC '14*, pages 75–80. IEEE.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. (2017). The first collision for full SHA-1. In *Proc. of CRYPTO '17*, pages 570–596. Springer.

Wang, H., Guo, Y., Ma, Z., and Chen, X. (2015). WuKong: A scalable and accurate two-phase approach to Android app clone detection. In *Proc. of ISSTA '15*, pages 71–82. ACM.

Wang, X. and Yu, H. (2005). How to break MD5 and other hash functions. In *Proc. of EUROCRYPT '05*, pages 19–35. Springer.

Zhan, X., Fan, L., Liu, T., Chen, S., Li, L., Wang, H., Xu, Y., Luo, X., and Liu, Y. (2020). Automated third-party library detection for Android applications: are we there yet? In *Proc. of ASE '20*, pages 919–930. ACM.

Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., and Gu, D. (2017). Embroidery: Patching Vulnerable Binary Code of Fragmentized Android Devices. In *Proc. of ICSME '17*, pages 47–57. IEEE.