Source Attribution of Cryptographic API Misuse in Android Applications

Ildar Muslukhov
The University of British Columbia
Vancouver, British Columbia, Canada
ildarm@ece.ubc.ca

Yazan Boshmaf
Qatar Computing Research Institute
Doha, Qatar
yboshmaf@hbku.edu.qa

Konstantin Beznosov
The University of British Columbia
Vancouver, British Columbia, Canada
beznosov@ece.ubc.ca

ABSTRACT

Recent research suggests that 88% of Android applications that use Java cryptographic APIs make at least one mistake, which results in an insecure implementation. It is unclear, however, if these mistakes originate from code written by application or third-party library developers. Understanding the responsible party for a misuse case is important for vulnerability disclosure. In this paper, we bridge this knowledge gap and introduce source attribution to the analysis of cryptographic API misuse. We developed BinSight, a static program analyzer that supports source attribution, and we analyzed 132K Android applications collected in years 2012, 2015, and 2016. Our results suggest that third-party libraries are the main source of cryptographic API misuse. In particular, 90% of the violating applications, which contain at least one call-site to Java cryptographic API, originate from libraries. When compared to 2012, we found the use of ECB mode for symmetric ciphers has significantly decreased in 2016, for both application and third-party library code. Unlike application code, however, third-party libraries have significantly increased their reliance on static encryption keys for symmetric ciphers and static IVs for CBC mode ciphers. Finally, we found that the insecure RC4 and DES ciphers were the second and the third most used ciphers in 2016.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Security requirements*;

KEYWORDS

Static Analysis, Source Attribution, Android, Cryptography APIs, Applied Cryptography

ACM Reference Format:
Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. 2018. Source Attribution of Cryptographic API Misuse in Android Applications. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3196494.3196538

1 INTRODUCTION

Adoption rates of smartphones have grown exponentially over the last decade. Android alone has enjoys more than 1.4 billion daily active users [7]. As a result, smartphone users started to accumulate large amounts of sensitive and private data on their personal mobile devices that requires confidentiality protection [28].

One way to achieve data confidentiality on smartphones is to enable filesystem-level encryption. For example, the current implementation of full-disk encryption in Android provides such a service as part of the storage IO stack [1]. Another approach is to implement custom, application-specific solutions using supported cryptographic APIs, or crypto APIs for short. For example, application developers can encrypt user data before storing it on the device or transmitting it over the network.

Unfortunately, neither one of the approaches is problem-free. Filesystem-level encryption relies on memorable passwords, which are often weak [16] and vulnerable to shoulder surfing [30]. Custom, application-specific data encryption often uses static encryption keys or initialization vectors (IVs) [20], which violates cryptographic notions of security, such as Indistinguishability Under Chosen-Plaintext Attack (IND-CPA) [19]. Recent advances in smartphone authentication, such as Apple TouchID, have partially addressed the problem of easy-to-guess passwords by making the use of longer, harder to guess secrets more convenient. The problem of crypto API misuse, however, is still far from being fully understood. While recent research shows that 88% of Android applications that use crypto APIs make at least one mistake [20], a study by Derr et al. [8] suggests that third-party libraries may be responsible for such mistakes. It is unclear, however, to which extent third-party libraries have contributed to the problem of crypto API misuse.

Problem. While third-party libraries used in Android applications misuse crypto APIs, it is unclear how prominent this problem is and what the resulting security implications might be. Attributing an API misuse to its source, be it an application code or a third-party library, is important since it is crucial for vulnerability disclosure and software patching, allowing security researchers and companies to address the root cause of the problem. In addition, it informs application developers how libraries that they are using or consider using consume crypto APIs.

Approach. The goal of this research is two fold: (1) to attribute crypto APIs misuse to its source, and (2) to study how misuse has changed between 2012 and 2016. To achieve that, we first obtained three datasets consisting in total of 132K Android applications from Google Play Store in 2012, 2015, and 2016. We then designed and implemented BinSight, an automated analysis system that: (1) identifies call-sites terminating in Java crypto APIs using static program slicing, (2) validates these call-sites against common rules in cryptography, and finally (3) attributes the misused call-sites to their sources using a heuristic-based third-party library detection technique.

Compared to existing program analyzers, such CryptoLint [20], AndroGuard [18], and Soot [25], BinSight offers three unique features: (1) it provides a reliable way to attribute a Java class identifier to its source, be it an application code or a third-party library, even if class renaming has been applied using Java bytecode obfuscation tools, (2) it scales to large Android applications that are usually excluded from analysis, and (3) it has a rich GUI for manual validation and analysis of API (mis)use.

Results. We used BinSight to analyze Android applications on a large scale. The analysis results suggest that third-party libraries are the major consumer of crypto APIs. In particular, for the applications collected in 2016, we found that 90% of call-sites that terminated in crypto APIs originated from 638 libraries. Furthermore, 79.4% of these libraries, a total of 507, were responsible for introducing crypto API misuses in 89.5% of the flagged applications. That is, if the library vendors fix crypto APIs misuse in 507 libraries they can reduce tenfold the number of Android applications with crypto API misuses.

Comparison of crypto APIs misuse between 2012 and 2016 revealed that, with some exceptions, both libraries and applications have improved. Unlike applications, libraries have significantly increased the use of static encryption keys and static initialization vectors (IVs) for ciphers in CBC mode. Furthermore, we found that both RC4 and DES ciphers, which are known to be insecure [23, 29], have become the second and third mostly used ciphers in 2016. At the same time, the popularity of 3DES has declined eight fold.

With BinSight's GUI tools, we manually analyzed the top-two libraries in each of the three datasets. The results of the manual analysis revealed that while all but one library have misused crypto APIs, the identified misuse cases had minimal or no impact on the actual security of the applications. For example, the Google Play SDK, which used a symmetric cipher in CBC mode with a static IV and a static encryption key, did so for obfuscation and not for confidentiality or integrity protection. We refer to such scenarios as a "functional" false positives, in which the crypto API is formally misused, but the misuse does not lead to insecurity.

Contributions. This paper makes the following contributions:

(1) A study of crypto APIs misuse in Android applications with source attribution. We show that 90% of call-sites to crypto APIs originate from third-party libraries, and source attribution based on package name is still efficient and effective, even if class name obfuscation techniques are employed.
(2) Trend analysis of crypto API misuse rates since 2012. The results of the analysis revealed that while applications and libraries improved in most aspects, e.g., the use of ECB mode and the use of static seeds for SecureRandom class, libraries became worse in the use of static encryption keys and static IVs. In addition, our results suggest that insecure RC4 and DES ciphers, have gained popularity, by becoming the second and third most used ciphers, and the use of 3DES, a more secure version of DES, declined by eight fold.
(3) We demonstrate that the previously used metric to measure crypto API misuse rates, i.e., the ratio of APK files, is highly biased towards popular libraries and might convey misleading results. We show how the ratio of call-sites with mistakes can provide additional insights into the state of crypto APIs misuse.
(4) We show that without a proper contextual knowledge, the static program slicing approach, employed by both the BinSight and CryptoLint tools, suffers from a significant ratio of functional false positives: a false positive that meets the formal definition of misuse, yet, does not introduce an exploitable vulnerability. We identified two instances of functional false positives: (a) use of symmetric cipher for obfuscation, i.e., no intention of confidentiality or integrity protection, and (b) edge case for ECB rule, in which a single plain-text block of random data is encrypted. Our analysis results provide evidence that the problem of functional false positives is substantial, considering that the the misuse in the most popular library from 2016 dataset was identified as functional false positive. That library was responsible for over a half (56% or 50,015) of APK files in 2016 dataset.
(5) Design and implementation of BinSight: an open source tool for automatic crypto APIs misuse detection that supports source attribution.[1]

2 RELATED WORK

The research community has paid significant attention to the (mis)use of cryptography in smartphone applications. For instance, Lazar et al. [26] studied Common Vulnerabilities and Exposures (CVE) that were reported between January of 2011 and May of 2014 and that were related to cryptography. The results of their analysis suggest that 83% of the CVEs were introduced by application developers that incorrectly used crypto APIs. To understand how this issue can be alleviated, Acar et al. [13] studied the usability of several cryptographic libraries. The results of the user study suggest that while making the crypto APIs simpler had its benefits, application developers still required proper documentation, code samples and certain features to be available for the library to be used properly.

Several researchers used static analysis methods and tools to analyze crypto APIs misuse in Android applications' binaries. For example, Fahl et al. [22] studied the misuse of asymmetric cryptography for SSL/TLS protocols, and certificates validation in particular. The analysis of 13,500 top free Android applications revealed that 8% of the analyzed applications misused SSL/TLS APIs, which made these applications potentially exploitable.

Egele et al. [20] developed the CryptoLint system that used static analysis to identify misuses of crypto APIs. In particular, based on IND-CPA notion of security [19], the authors defined six rules of secure use of crypto APIs for symmetric ciphers, password based key derivation function (PBKDF), and secure random number generators. Their analysis revealed that 88% of Android applications that use crypto APIs violated at least one rule. Similarly to the CryptoLint study, we focus on the same set of rules (reproduced in Section 3), while introducing source attribution to the analysis

[1]The project can be found at https://github.com/iim/binsight. Due to large size of the datasets, we will share them upon request. Contact first author for your requests.

pipeline. In addition, we extended the original dataset of the CryptoLint study by adding newly collected applications from 2015 and 2016, and we also made the BinSight tool available as open source.

Finally, Derr et al. [8] studied how promptly application developers adopt new versions of the libraries, especially when there is a known vulnerability in the library. While doing so, the authors also evaluated the six rules defined in the CryptoLint study for the identified libraries. Unsurprisingly, the results of the analysis revealed that libraries violated these rules too. In contrast, we focused on studying violation of the rules of using crypto APIs from specific source, i.e., a library or the application itself.

To summarize, while previous research has looked into either libraries or Android applications as a whole, the problem of attributing the source of a crypto APIs misuse is still not addressed. Attributing crypto APIs misuse to its source is crucial for several reasons. First, one needs to clearly identify the responsible party for fixing the bug. Second, identifying the source of a misuse allows researchers to reduce over-counting of bugs, by identifying ones that originate from libraries. In addition, by being able to analyze binaries, the BinSight tool allows application developers to get an insight into how a library (mis)uses crypto APIs. This allows them to make an informed decision whether or not they want to use this library in their applications.

3 COMMON RULES IN CRYPTOGRAPHY

Similar to related work [20, 31], we investigated whether an Android application that uses Java cryptographic APIs achieves a cryptographic notion of security, IND-CPA security in particular.[2] Accordingly, we use the rules originally defined by Egele et al. [20], and flag all applications that violate any one of them. While it is safe to flag these applications as insecure, we note that a flagged application might be using the APIs for purposes other than protecting data confidentiality or integrity, as suggested in §2. In the rest of this section, we partially reproduce definitions of these rules. For a detailed description, we refer readers to the original CryptoLint paper [20].

3.1 Symmetric key encryption

In block ciphers, a mode of operation defines security properties the cipher would provide, such as confidentiality. A popular mode is electronic codebook (ECB), which is not IND-CPA secure. The major problem with ECB mode is that identical messages encrypt to identical ciphertexts, which represents an information leak that is often intolerable. Still, ECB mode is commonly considered secure if the message fits into a single cipher block and each messages are unique. Therefore,

Rule 1. *Do not use ECB mode for encryption.*

Another popular mode of operation is ciphertext block chaining (CBC), where each block of plaintext is XORed with the previous block of ciphertext, before being encrypted by the block cipher. The first block of plaintext is XORed with an initialization vector (IV). Using a constant IV will result in a deterministic, stateless cipher, which is not IND-CPA secure. Thus,

Rule 2. *Do not use a constant IV for CBC mode.*

Any symmetric encryption scheme, defined using a block or a stream cipher, should not reveal its key. If the key is hard-coded into a publicly-available application as a constant, then the key is not private, and so the resulting encryption does not provide confidentiality and integrity protection. Symmetric encryption schemes commonly assume a random key. Accordingly,

Rule 3. *Do not use constant encryption keys.*

3.2 Password-based encryption

User-created passwords are often weak and vulnerable to password guessing attacks. Password-based encryption (PBE) schemes significantly increase the costs of such attacks. They achieve this by concatenating the password with a salt and applying multiple iterations of a cryptographic hash function, typically using a key derivation algorithm. The salt and the iteration count entail a multiplicative increase in the work required for a guessing attack. Using a constant salt is equivalent to not having a salt at all and using fewer than 1,000 iterations makes password guessing attacks practical. We note that this threshold for the iteration count is the minimum value suggested by RFC 2898 [24]. Hence,

Rule 4. *Do not use constant salts for PBE, and*

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

3.3 Random number generation

Android provides an API to a seeded, cryptographically-strong pseudo-random number generator (PRNG) via SecureRandom class. This PRNG is designed to produce non-deterministic output, but if seeded using a constant value, it will produce a constant, known output. If such a PRNG is used to derive keys, the resulting keys would not be random, making the encryption insecure. As such,

Rule 6. *Do not use a constant to seed SecureRandom.*

4 CRYPTOGRAPHY IN ANDROID

There are various reasons to use cryptography in Android applications. We now give an overview of the application ecosystem in Android, focusing on packaging and Java run-time. We then present a brief introduction to the use of cryptography in Java.

4.1 Android applications ecosystem

Android applications are authored as either native C/C++ or Java source code. We only consider applications that are written in Java, because Java has had stable crypto APIs since the release of Java 1.4 in 2002. An Android Java application is compiled to Dalvik executable (DEX) bytecode. The application is packaged into an APK file with all required resources, such as images or third party libraries. The APK file is then uploaded to Google Play Store, and when a user installs the application, the APK file is downloaded and installed on their device.

[2]Indistinguishability Under a Chosen-Plaintext Attack (IND-CPA) is a basic requirement for most provably secure public key crypto systems. Informally, an encryption scheme is called IND-CPA secure if an attacker is unable to distinguish pairs of ciphertexts with a probability better than that of random guessing [19].

Name	Number of APKs	Sampling	Year
R16	117,320	Random	2016
R12	10,990	Random	2012
T15	4,280	Top-100	2015

Table 1: Summary of used datasets

Even though a DEX bytecode is compiled from Java, the Dalvik virtual machine (DVM) is considerably different from the Java virtual machine. Unlike Oracle Java virtual machine, which is stack-based, DVM is register-based, with a dedicated assembly language called Smali. However, it is possible to convert a DEX bytecode to an Oracle Java bytecode with Dex2Jar tool [6], albeit with some limitations, such as inability to decode specific classes. We note that DVM was recently replaced by Android runtime (ART), which translates the DEX bytecode into the CPU's native instructions for faster execution.

4.2 Java cryptography

Android provides a rich execution framework that offers access to various sub-systems, including Java cryptography architecture (JCA). The JCA standardizes how developers make use of many cryptographic algorithms by defining a stable API. Accordingly, a cryptographic service provider (CSP) is required to register with the JCA in order to provide the actual implementation of these algorithms. This abstraction allows developers to replace the default CSP, which is BouncyCastle [4] in Android, with a custom CSP that satisfies their requirements. For example, SpongyCastle [5] is a popular third-party CSP that supports a wider range of crypto algorithms.

Symmetric and asymmetric encryption schemes are accessible to developers through the `Cipher` class, as illustrated in Listing 1. To use a specific encryption scheme, the developer provides a transformation as an argument to the `Cipher.getInstance` factory method. A transformation string specifies the name of an algorithm, a cipher mode, and a padding scheme to use in the `Cipher` object [12]. In Listing 1, the returned cipher instance uses AES in CBC mode with *PKCS#5* padding. Only the algorithm name is mandatory, while the cipher mode as well as the padding scheme are optional. Unfortunately, all CSPs default to ECB mode of operation if only the cipher name is specified, which is insecure [11].

Listing 1: Simplified symmetric key encryption in Java

```
// values of iv and key should be randomly generated
public byte[] encrypt(byte[] iv, byte[] key, byte[] data) {
  IvParameterSpec iv_spec = new IvParameterSpec(iv);
  SecretKeySpec key_spec = new SecretKeySpec(key, "AES");

  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
  cipher.init(Cipher.ENCRYPT_MODE, key_spec, iv_spec);

  return cipher.doFinal(data);
}
```

5 DATASETS

As summarized in Table 1, we collected and analyzed three datasets, with 132,590 APK files in total. R12 is a subset of the CryptoLint dataset with 10,990 APK files. The original CryptoLint dataset had 145,095 APK files and was collected between May and July of 2012 [20] by crawling Google Play marketplace. First, the authors of CryptoLint excluded all APK files that did not use crypto APIs. Second, the authors also excluded all APK files that had all Crypto API calls originating from 11 white-listed libraries. This resulted in a set of 15,134 APK files. The CryptoLint tool, however, failed to analyze 3,386 files from this set and 758 files were lost since 2012, resulting in 10,990 APK files in the R12 dataset. Considering that 758 lost files are a random sample of the set that was presented in the CryptoLint report [20], such loss does not have a significant impact on our results.

The R16 dataset was collected in May of 2016 with help of Sophos. To select APK files in the R16 dataset we first generated a random sample of 120,000 APK files that were available on Google Play market at that time and then downloaded that set through Sophos servers. Unfortunately, 2,680 files got corrupted during the downloading process, leaving us with 117,320 APK files. During the analysis, we discovered that applying the same white-listing approach that the authors of CryptoLint used did not introduce practically and statistically significant difference.

Finally, the T15 dataset includes the Top 100 Android applications in each category from June 2015. For this dataset, a list of the Top 100 applications in each category was first obtained through Google Play store APIs. Then each APK file was separately downloaded with the ApkDownloader tool [10]. The downloading process was performed between June 13–28, 2015. As 20 applications were removed from the Google Play Store before we were able to download them, the final size of the dataset is 4,280.[3] We compared T15 to R16 only for additional insight into differences between random and top applications.

6 CRYPTO API LINTING WITH BINSIGHT

At a high level, the rules defined in §3 represent temporal properties that can be validated using automated program analysis in a task known as linting [21]. While previous research has proposed various linters for Android crypto APIs [20, 31], they suffer from various limitations. In particular, the state-of-the-art linter CryptoLint is not available as open source and was unable to sanalyze more than 20% of APK files [20]. In addition, none of these tools provide any code navigation, which is valuable for manual in-depth analysis. Finally, existing tools do not support attribution of the source of misuse, i.e., by using these tools one cannot tell whether a misuse is due to an application code or a third-party library.

To overcome these limitations, we developed BinSight based on technical description of the CryptoLint [20]. In comparison to CryptoLint, BinSight also provides a rich, graphical UI for manual analysis of an APK file and source attribution. To overcome limitations of CryptoLint study, we introduced two new stages to the APK analysis pipeline. The technical details of linting pipeline are provided in Appendix 11.1. In what follows, we focus on explaining our approch to the source attribution.

[3]Due to large size of the T15 and R16 datasets we cannot make them available online, but can share upon request. For the R12 dataset we refer readers to contact the authors of the CryptoLint study.

	Number of APKs			
Name	Total	Unique	Dups	Crypto?
R12	10,990	10,222	768 (7%)	10,222 (100%)
R16	117,320	115,683	1,637 (1.4%)	95,775 (82.8%)
R16*	117,320	115,683	1,637 (1.4%)	93,994 (81.3%)
T15	4,280	4,067	213 (5%)	3,645 (89.6%)
Total	132,590	129,972	2,618	109,642

Table 2: Summary of duplicates and crypto APIs use in all three datasets. R16* is a subset of R16 with CryptoLint libraries whitelisting applied.

	Class identifier renaming level				
	None	Class	Partial	Full	Total
R16	509,643	203,447	106,091	21,279	840,460
	60.64%	24.21%	12.62%	2.53%	100%
R12	78,883	14,513	6,882	2,002	102,280
	77.12%	14.19%	6.73%	1.96%	100%
T15	26,821	12,907	3,620	1,804	45,152
	59.40%	28.59%	8.02%	3.99%	100%

Table 3: Obfuscation analysis of class identifiers.

6.1 Attribution

After the linting stage, every call site that terminates in a crypto API is attributed to its source, which could be the application code itself or a third-party library. Our attribution approach relies on package names of classes that contain call to crypto APIs, and cross-references them with an exhaustive list of third-party libraries in our datasets. The attribution has to handle obfuscated package names, in order to correctly map call sites to libraries. This is done in the following two steps.

6.1.1 Obfuscation analysis. Although de-obfuscating Android applications has been recently studied [14, 27], the underlying techniques, while effective for manual forensics, are inefficient for analyzing applications on a large scale. Moreover, it is unclear how prominent the use of obfuscation is in the real-world, especially in the classes that use crypto APIs. To automatically detect the level of obfuscation, we developed a simple, rule-based classifier that identifies whether a given package name is fully obfuscated or not. The heuristic rules are provided in Appendix 11.2. If the package is not fully obfuscated, we found that in 99% of the cases one can still use its name to identify the library it belongs to. We show in §7 that less than 2.5% of package names in R16 were fully obfuscated, requiring sophisticated de-obfuscation techniques. We refer the reader to the implementation of the *UseCase* class in the *APKInsight.Logic.Analysis.Data* name space of BinSight project for more details on the implementation of heuristic rules we developed.[4]

6.1.2 Third-party library detection. As mentioned above, almost all call sites that terminated in crypto APIs correspond to package names that were identifiable. We labeled the exceptions, which are fully obfuscated package names, as "obfuscated," meaning that we were unable to attribute the source of their corresponding call sites. For the remaining majority, we decided whether a package name corresponds to an "application," a "library," or a "possible library", by searching an exhaustive list of package names for libraries and potential libraries. We manually compiled this list by inspecting all unique package names across the datasets, as described in §7. Our approach complements Derr et al. [8] technique, which relies on a list of library signatures extracted from a database of third-party SDKs [8].

[4]https://github.com/iim/binsight

7 MEASURING CRYPTO API MISUSE

This section presents and discusses the result of the analysis of 109,642 APK files that had at least one call to crypto APIs. To the best of our knowledge, this is the largest dataset analyzed for crypto APIs misuse (e.g., the CryptoLint study is based on the analysis of 11,748 APK files). First, we discuss duplicates, obfuscation detection and source attribution for each of the datasets. Then we present the overall statistics on crypto APIs misuse. Afterwards, we proceed with the analysis of each rule separately. In our analysis, we compare R12 to R16 in order to understand what have changed between 2012 and 2016, and T15 to R16 in order to understand how a top application differs from a random one.

For each comparison we conducted Chi-square test to see whether the found difference was statistically significant with 99% confidence. In what follows we discuss only statistically significant results and all figures show 99% confidence interval whiskers.

7.1 Preprocessing

Unsurprisingly, every application in R12 made at least one call to crypto APIs, confirming the analysis and the white-listing performed by the authors of CryptoLint [20]. Interestingly, while they found that only 10.4% of the applications called crypto APIs, in R16 and T15 we found that 83% and 90% of the applications used crypto APIs. Such a significant increase in use of crypto APIs in Android applications can be attributed to many factors, including the white-listing the authors of CryptoLint applied or increased necessity to protect users' data.

Our analysis revealed that while all datasets contained duplicates, R12 had the largest ratio (7%). We removed all duplicates from the analyzed datasets. The summary of the datasets after de-duplication is shown in Table 2.

Unlike CryptoLint, BinSight was able to disassemble and analyze all but six of the 109,642 APKs. This represents a significant improvement over CryptoLint, which failed to analyze 3,386 APK files (23% of the analyzed set) due to technical problems.[5] BinSight completed analysis in about 14 days on a dual Xeon CPU with 128GB RAM, i.e., processing about 7500 APK files a day, which suggests that BinSight is not only robust, but also scalable.

7.2 Linting and attribution

[5]According to CryptoLint authors, there were two major problems (a) the tool did not finish analysis within 30 minutes, and (b) the analysis infrastructure ran out of memory.

		Source (%)			
	Call sites	Libs	Apps	Libs?	?
R16	840,460	90.7	4.9	1.9	2.5
R12	102,280	79.5	14.5	4.0	2.0
T15	45,152	80.6	10.7	4.7	4.0

Table 4: Attribution of cryptographic API call sites.

7.2.1 Obfuscation analysis. As noted in §6, it is unclear how prominent the use of obfuscation is, and class identifier renaming (CIR) [8] in particular. Accordingly, we analyzed the three datasets to quantify CIR in the real-world [8]. We limited the analysis to only those classes that have at least one call site to crypto APIs. While doing so served our needs, our results on the prevalence of obfuscation should not be considered as a generalization to all Android applications.

There are different levels at which CIR can be applied by an obfuscator like DexGuard. For instance, for class *com.domain.package.Class*, an obfuscator might not change the identifier, rename the class only, rename the class and partially its package, or rename the whole class identifier. For the first three levels, we can map the class to a library or an application, if the package name has an identifiable prefix. As for the fourth level, we cannot use the package name for source attribution.

Unlike previously published research, e.g., LibScout [8], we did not aim to detect different versions of the same libraries. Our goal was simpler. That is, we aimed to tell if a class belongs to a specific library or to an application. To assess the reliability of using package names for source attribution, we first automatically compiled a list of all unique class identifiers that call crypto APIs. We then semi-automatically inspected the list in order to determine if the identifiers were obfuscated. If in doubt, we used BinSight GUI to inspect the internals of a class and its source file name, when that was available.

To our surprise, the analysis revealed that *using package names for source attribution is a reliable method for source attribution.* In particular, for applications in R16 we were able to identify the source for 97.5% classes that made calls to crypto APIs. The results of the analysis for all three datasets are provided in Table 3.

7.2.2 Third-party library detection. We classified package names into one of the four categories: applications (apps), libraries (libs), possible libraries, and obfuscated. We now describe how we performed this classification. First, we assigned all package names that have been fully obfuscated to category *obfuscated.* We then assigned all package names that were found in a single application to category *applications.* For the remaining packages, which were found in two or more applications, we ranked them based on how many applications used them in each dataset, and then performed manual inspection in a decreasing order of the rank. In particular, for each package name, we labeled the package name as a library if we were able to find library source or website. Furthermore, if previous has failed we then used BinSight's GUI for manual inspection to verify if the package under investigation belongs to a library. We stopped manual analysis once we identified enough package names to cover 95% of the call sites. We assigned the remaining unclassified package names to the *possible libraries* category.

In total, we manually analyzed 12,165 package names from the three datasets, out of which 3,622 (29.7%) belonged to libraries. Overall, we identified 638, 260, and 265 libraries in R16, R12 and T15, respectively. This suggests that BinSight significantly improved upon CryptoLint in terms of libraries detection[6].

Our analysis based on source attribution revealed that the libraries were responsible for the majority of calls to crypto APIs in all three datasets, as summarized in Table 4. Even more, 79.5% of all calls to crypto APIs in the R12 dataset originated from 260 libraries. While the authors of CryptoLint study did white-list 11 libraries, analysis with BinSight allowed us to identify the remaining 249 libraries, which accounted for 79.5% of the calls to crypto APIs in R12. This suggests that BinSight significantly improves the accuracy of the results reported in [20].

To this end, we showed that (a) one can reliably use package name for source attribution, since it covers 97.5% of the calls to crypto APIs, (b) libraries are the major contributor to crypto APIs calls and should be properly identified, and (c) previously published research (i.e., [20]) has missed more than 200 libraries, which suggest that its results suffer from the over-counting problem.

7.3 Crypto APIs misuse

In what follows, we present the main findings on crypto APIs misuse rates across all source categories. We begin with the results of the analysis on overall misuse rates across all rules, i.e., at least one rule is violated. Afterwards, we proceed with analysis of misuse rates for each rule separately. For brevity, we omit results for *possible libraries* and *obfuscated* call-sites source categories.

During the analysis we observed that the ratio of APK files with misuses, the metric used by CryptoLint study, has its limitations. In particular, while such measure provides an intuition on the overall share of APK files with crypto APIs misuses, it is heavily biased towards libraries, especially the popular ones. That is why in our analysis, in addition to the ratio of APK files with misuses, we used the ratio of crypto API call-sites that make a mistake. The main reason for measuring this ratio is due to the fact that it is trivial to separate calls to crypto APIs based on source. Such separation allows clearer understanding of trends within each source.

For both of the aforementioned metrices, we report results for all sources combined and separately. The ratio of APK files with misuses per category is computed against the total number of APK files in the dataset. Because an APK file might contain misuses from various sources, the sum of ratios for all four categories will be equal to "All" category. The ratio of call-sites with mistakes, however, is assessed against the total number of calls that originate from that category.

7.3.1 Overall crypto APIs misuse rate. The ratios of APK files with at least one violation of the rules per category are shown in Figure 1(a). Unsurprisingly, our results for R12 sub-set were in-line with previously reported, i.e., 95% in our study and 88% in CryptoLint study [20]. We attribute the difference to two factors: (a) we removed 7% of APK files from the R12 dataset, as they were

[6]CryptoLint authors white-listed 11 libraries

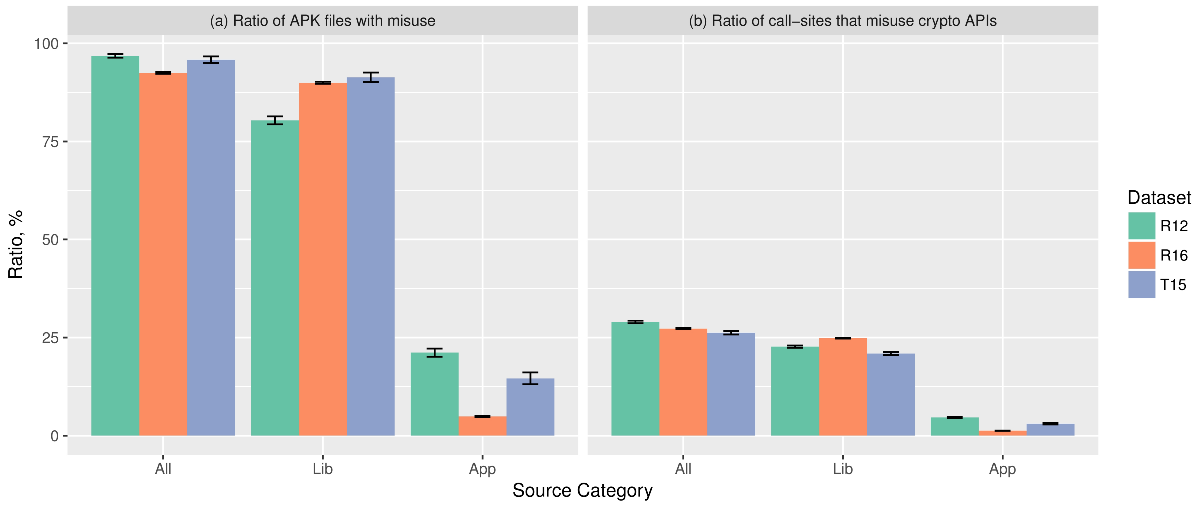

Figure 1: Ratio of APK files and call-sites that violated at least one of the crypto APIs use rules, per dataset. “All” category results are based on all call-sites together, without considering the source (i.e., library or an application). *Libs* and *Apps* represent APK files or call-sites that originate from libraries or applications.

duplicates, and (b) 768 APK files from the original the R12 dataset were lost. As expected, we found that the white-listing approach used in the CryptoLint study reduced the ratio of APK files to which libraries have introduced misuses. Yet, it did not have any impact on the call-sites ratio, as shown in Figure 1(b).

Overall, we found that since 2012 the ratio of APK files with at least one misuse has decreased from 94.5% to 92.4%. At the same time, the overall likelihood of a call-site to crypto APIs to made a mistake remained around 28%, i.e., on average *one out of four calls to crypto APIs makes a mistake.* Per category analysis, however, showed that while libraries have increased the ratio of APK files they introduced a misuse of crypto APIs to (from 80% to 90%), the likelihood of a call-site to make a mistake from libraries did not show a statistically significant change. The lack of change is probably because the number of libraries has increased (from 260 in R12 to 638 in R16).

Unlike libraries, applications have improved in both the ratio of APK files and the likelihood of a call-site that make a mistake. In particular, the ratio of APK files decreased from 21% to 5% and the ratio of call-sites from 31.8% to 27.7%. Although, the increase in the total number of libraries might have also contributed to the decrease in the ratio of APK files applications contribute misuses to. Comparing T15 with R16 revealed that applications were introducing crypto APIs misuses to a larger share of APK files (5% in R16 vs 14.6% T15). This difference, however, could be attributed to the fact that T15 had fewer libraries (265 in T15 compared to 638 in R16).

7.3.2 Rules 1 – 3: Symmetric key encryption.

The overall use of ECB mode for symmetric ciphers has significantly decreased since 2012, as shown in Figure 2(a) - Rule 1. The number of APK files with cases of ECB mode use has dropped from 77% in R12 to 30% in R16. Similarly, the ratio of relevant call-sites has dropped from 53% to 29% (see Figure 2(b) - Rule 1). Source attribution revealed that this decrease can be mainly attributed to improvements in libraries. In particular, while applications decreased the ratio of relevant call-sites that use ECB mode from 63% to 47%, libraries have reduced this ratio from 52% to 26%, i.e., a two fold improvement. Comparison of the T15 and R16 datasets revealed that a randomly selected application is less likely to use ECB mode than a top application.

Despite the positive outlook on the use of ECB mode, we found that there was a statistically significant increase in the use of static IVs. In particular, since 2012 the ratio APK files that use symmetric ciphers with a static IV in CBC mode has increased from 32% to 96% (see Figure 2 - Rule 2). The ratio of relevant call-sites has increased from 31% to 71%. Libraries were the main source of the increase. Applications, at the same time, have reduced the ratio of call-sites that violated Rule 2. A comparison of the T15 and R16 datasets did not reveal any practically significant changes, i.e., both of these datasets were comparable.

By 2016, the ratios of APK files and call-sites to symmetric ciphers that use static encryption keys have increased (see Figure 2 - Rule 3). In particular, the ratio of APK files that violate Rule 3 increased from 70% to 93%, and the ratio of call-sites that use symmetric cipher with static key increased from 45% to 57%. Both, applications themselves and libraries, have become worse. Although, one might say that the ratio of APK files for applications have decreased, this, however, was due to the increase in the number of libraries and the ratio of all calls libraries make. This provides evidence that using the ratio of APK files with misuses is biased towards libraries.

In addition, we extracted the top-5 used symmetric ciphers from each dataset, as summarized in Table 5. We observed two troubling patterns. First, we found that the RC4 cipher has made it to the top-3 used ciphers in both T15 and R16, even though it is considered insecure [23] and security community has suggested to remove it from cryptographic libraries [29]. Second, the results revealed that

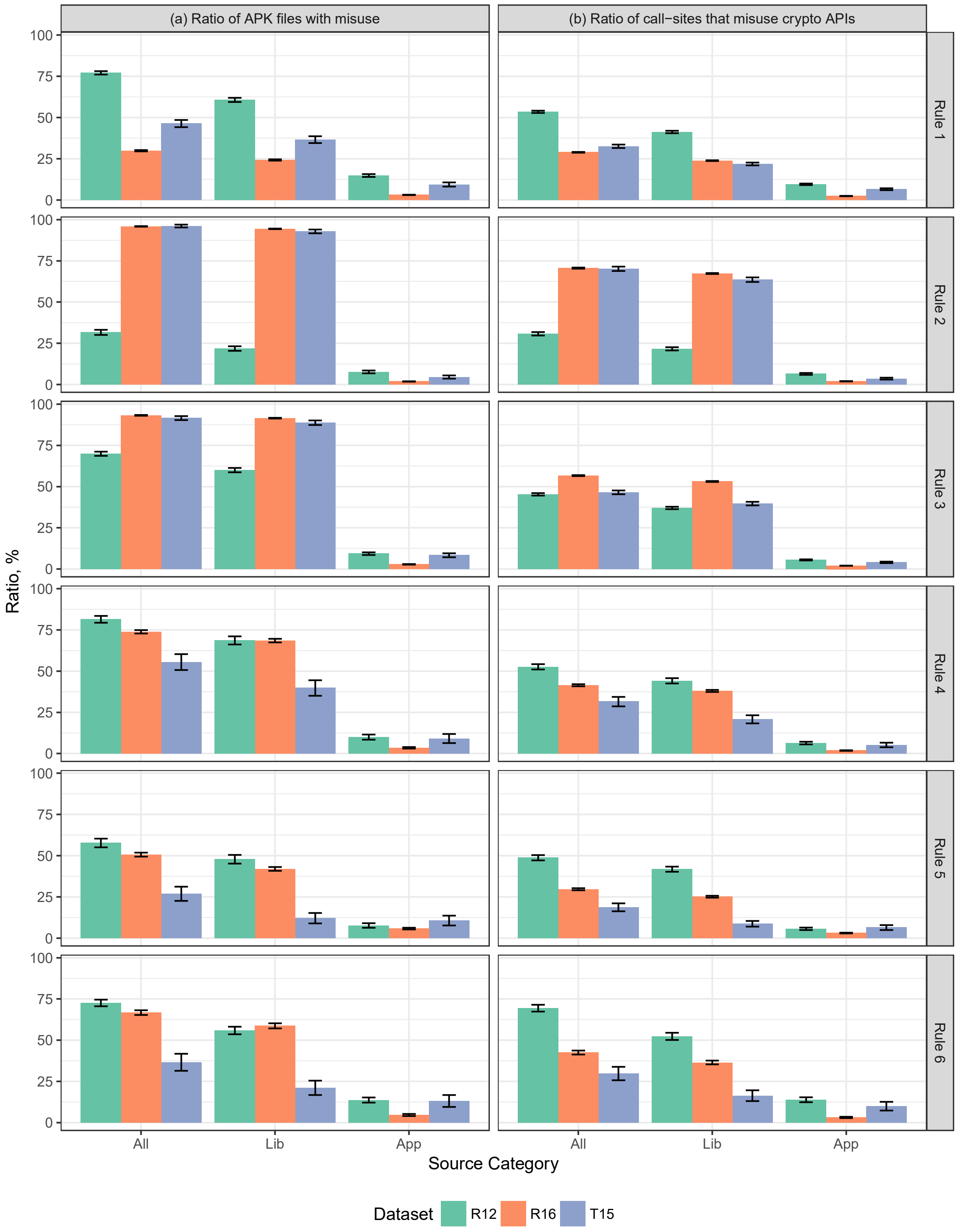

Figure 2: The ratios APK files (side a) and call-sites (side b) that misuse crypto APIs. "All" group presents data for all categories together. Each sub-graph breaks down all call-sites into four identified sources, such as libraries and applications. Whiskers represent 99% confidence intervals.

		Cipher (%)					
	Call sites	AES	DES	3DES	RC4	Blowfish	Others
R16	251,021	64.4	14.3	1.1	2.1	0.9	17.2
R12	31,192	58.9	19.0	8.8	0.4	1.9	10.9
T15	14,105	67.8	9.8	0.8	1.1	0.8	19.7

Table 5: The top-5 ciphers used in Android applications.

while DES remained second most used cipher, the popularity of 3DES, a more secure version of DES, has decreased eight folds.

To summarize, while the the popularity of ECB mode has significantly decreased, the use rates of static IVs for CBC mode and static encryption keys have increased. In addition, insecure ciphers, namely DES and RC4, were the second and third most used in 2016.

7.3.3 Password-based encryption. The rates of misuse of PBKDF have overall decreased for both static salts (Rule 4) and the number of iterations (Rule 5), as shown in Figure 2 - Rule 4. In particular, the ratio of APK files that used static salts for PBKDF has decreased from 81% to 74%. The ratio of APK files that used fewer than 1,000 iterations decreased from 58% to 51%. The ratio of calls to relevant crypto APIs that violate either rule 4 or 5 has also decreased (as shown on Figure 2 - Rule 5). Source attribution analysis showed that both libraries and applications have improved.

Comparison of T15 and R16 showed that, on average, 19% and 24% less APK files from T15 violated Rules 4 and 5, respectively. Analysis based on source attribution revealed that this effect was mainly due to improvements in libraries in the T15 dataset and not applications.

To summarize, in 2016 dataset, both applications and libraries improved their use of PBKDF for PBE.

7.3.4 Random number generation. The use of static seed values for `SecureRandom` has significantly decreased since 2012 (Figure 2 - Rule 6). In particular, while the ratio of APK files that used a static seed for `SecureRandom` class has dropped from 73% to 67%, the ratio of relevant call-sites with static seed value decreased to 43%, from 69%. Analysis of call-sites revealed the likelihood of using static seed in libraries has significantly decreased (from 72% to 42%). Comparison of the T15 and R16 datasets revealed that while libraries in T15 significantly outperformed those in R16, applications in T15 had higher violation rates than those in R16.

To summarize, both applications and libraries improved when it comes to the use of `SecureRandom`.

7.4 The impact of third-party libraries

The results of source attribution analysis revealed that R12, R16 and T15 contained 260, 638 and 265 libraries respectively, 222 (85%), 507 (79%) and 198 (75%) of which violated at least one crypto APIs use rule. These libraries with violations were the only source of misuses for 6,932 (70%), 79,207 (89.5%), and 2,629 (75.3%) of APK files in R12, R16, and T15 respectively. To this end the BinSight system allowed us to improve upon CryptoLint results by identifying originally missed 249 libraries (out of 260) in the analysis of the R12 dataset.

Another important factor to consider for the analysis of libraries is their popularity. That is, a popular library with a misuse will

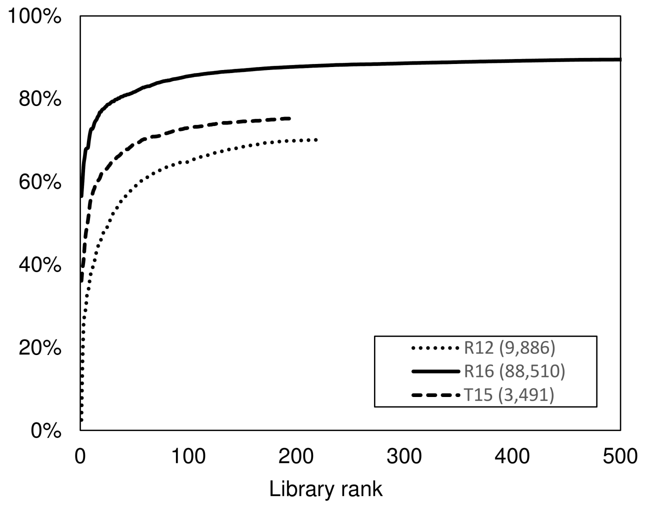

Figure 3: Proportion of APK files that would become free from crypto APIs misuse, depending on the number of fixed top ranked libraries. The legend shows the total number of applications that had at least one misuse in the corresponding dataset. We identified 222, 507 and 198 libraries with misuse in the R12, R16 and T15 datasets, hence, the end of the corresponding curves.

impact a significantly larger subset of APK files. To understand how the popularity impacts the misuse rates we proceeded with the following analysis: we measured the number of APK files that would be misuse-free if one starts fixing libraries, starting with the most popular ones first. Figure 3 shows this impact for each dataset. In particular, by fixing the top most library in R16, one would make 50,015 APK files misuse free (or 56% of all APK files with misuses), and by fixing all 507 libraries with crypto APIs misuse, one would fix 79,207 APK files (or 89.5% of APK files with misuses).

7.5 In-depth analysis of the top libraries

Considering that the top libraries are responsible for a large portion of flagged APK files, e.g., the top most library in R16 was responsible for 56% of the APK files, we conducted an in-depth manual analysis of the top-2 libraries, shown in Table 6, from each dataset. This resulted in five libraries analyzed, since one library was in top-2 for two datasets. The main objective of the manual analysis was to understand the purpose for the use of cryptography and the security implications of misuse.

				Rank in dataset		
Company	Library	Package	Violated rules	R16	R12	T15
Google	Play SDK	com.google.android.gms.internal	2, 3	1	–	1
Apache	HTTP Auth	org.apache.http.impl.auth	1	2	3	5
InMobi	Advertising	com.inmobi.commons.core.utilities.a	–	3	–	2
Google	Advertising	com.google.ads.util	3	38	1	36
VPon	Advertising	com.vpon.adon.android.utils	1, 3	–	2	–

Table 6: Top-2 libraries that use Java cryptographic APIs. Empty values mean the library was not found in the dataset.

7.5.1 Google advertisement. This library was the top library in the R12 dataset, and in the top 40 in the T15 and R16 datasets. This library provides advertisement services to applications. It makes use of data encryption API in `AdUtil` class, located in *com.google.ads.util*. The implementation uses static key for encryption (i.e., violates Rule 3), which is hard-coded in `AdUtil` class. The encryption function receives plaintext as a string and returns cipher text, also as a string. The encryption uses AES cipher in CBC mode with PKCS5Padding. The encryption function is later used to encrypt a string representation of user's location, before it being sent back to Google's servers. Considering that the communication happens over HTTPS protocol, the use of a static key does not impact confidentiality in the presence of a network attacker. In addition, we found that in R16 and T15, this library has significantly changed. In particular, the newer version did not use encryption anymore. The structure of the library got significantly simplified as well. There were, however, several applications in both T15 and R16, where the old version of the library was used. Interestingly, these applications were relatively recently updated (2 – 3 months prior the data collection of T15 and R16). This observation confirms is in line with the findings from a recent study [8] that suggests that application developers are usually slow to adopt new versions of the libraries.

7.5.2 VPon advertisement. This library is also an advertisement library, present only in the R12 dataset. It uses a cipher to encrypt and decrypt data. All identified call-sites were located in `CryptUtils` class, located in *com.vpon.android.utils* package. This library violated two rules, the use of ECB mode (rule 1) and static encryption key (rule 3). `CryptUtils` class exposes two types of encryption functions, one that uses *javax.crypto.SealedObject* as an input for encryption, and one that accepts key and data as a string and returns a string as a result. The functions that work with `SealedObject` are used to encrypt requests that are sent back to the server and decrypt responses from it. This suggests that the static key is shared between the library and VPon's servers. The requests are sent both, over HTTP and over HTTPS. Unfortunately, we were not able to understand exactly which data are sent over which protocol. The second function, based on strings as input and output, is only used to decrypt obfuscated string literals. Decrypting string literals in Android applications is a common obfuscation technique. To summarize, this library violates two rules (use of ECB mode and use of static key) to communicate with the advertisement server and to obfuscate data.

7.5.3 Apache library. This library allows applications to communicate over HTTP and HTTPS protocols. The library, was the only library in top 5 in all three datasets. It called crypto APIs in multiple locations, but one specific call site, which used ECB mode, drew our attention. In particular, the ECB mode is used in the implementation of a suit of NT Lan Manager (NTLM) authentication protocols, which are commonly used to authenticate over HTTP(s). This protocol, by design, uses DES cipher in ECB mode (i.e., violates rule 1), to implement challenge response validation. It, however, encrypts only a single cipher block (i.e., 8 bytes) consisting of a random challenge, which creates cipher-text indistinguishable from each other. This case is an example of functional false positive, i.e., the use of ECB mode in such scenario (single block of random data). It, however, is prone to other attacks, such as exhaustive search of the encryption key, due to the use of a insecure cipher, that is DES.

7.5.4 Google Play SDK. This library provides services of Google Play platform, such as In-App purchases or authentication with Google accounts. It was the top most library in both T15 and R16, and was absent from the R12 dataset. The library violated two rules, the use of static IV and static keys (rules 2 and 3). Interestingly, this library implemented decryption function only, that accepts an array of bytes as a key and a string as a cipher-text and returns the plain-text as a byte array. We found that the key is hard-coded as a property in a static class, which is located in *com.google.android.gms.internal* package. The same static class contains all the cipher-texts that get decrypted. Further analysis revealed that one of the cipher-texts was actually an encrypted DEX file, which upon decryption (about 3Kb in size) was loaded into application's space through Java Reflection API. The remaining cipher texts were string literals that identified properties and functions of the dynamically loaded class. It is clear, that this is a case of obfuscation, hence, a functional false positive.

7.5.5 InMobi advertisement. This library (second most popular in T15) allows applications to show In-App advertisements. The call-sites to Cipher facilities were found in `InternlSDKUtil`, located in *com.inmobi.commons.internal* package. This class uses AES cipher in CBC mode with PKCS7 padding. It also sends to the server a symmetric key encrypted with RSA cipher. We found that this library, similarly to VPon, uses encryption facilities to encrypt communications with their back-end server. Interestingly, we saw the use of both HTTP and HTTPS protocols for the communication to the same host name, thus, it is unclear why the library developers had not switched all communications to HTTPS. This library generates an encryption key once, stores it in `SharedPreferences` and then reuses it on all sub-sequent communications. Formally, InMobi's implementation does not violate any of the IND-CPA rules related to the cipher.

7.6 The impact of third-party libraries revisited

The results of in-depth analysis of the top libraries revealed that the current approach for identifying crypto APIs misuses in Android applications might be suffering from a significant ratio of functional false positives. We classify a misuse case as a functional false positive if the actual use of the crypto APIs was not meant to provide integrity or confidentiality protection. For example, while Google Play SDK violated rules 2 and 3, it did so for obfuscation purposes only. Another limitation of the current approach is missing certain edge cases, e.g., encryption of a single block of random data in ECB mode. Such cases, however, significantly inflate the misuse rates (e.g., the impact of Apache library on overall misuse rates), and thus, convey a wrong state of actual misuse of cryptography in Android applications. Future research should focus on expanding BinSight's ability to classify if cryptographic APIs are used for obfuscation purposes.

8 DISCUSSION

The results of our analysis revealed that both applications and libraries decreased their reliance on ECB mode for symmetric ciphers. Libraries, however, have significantly increased the use of static IVs and static encryption keys. This suggest that while application developers tried to move away from insecure ECB mode, they failed to do so properly. The failures to adopt secure encryption might be explained by the lack of understanding or incomplete documentation [13]. Another possible factor is the introduction of a warning message into the Android Studio after the CryptoLint study was conducted. The warning message highlights the insecurity of ECB mode (*"...because the default mode on android is ECB, which is insecure."*). To our surprise, we found that the Crypto Stack-Exchange[7] is full of invalid suggestions on how to fix this warning message.

The use of PBKDF has also improved since 2012. In particular, both applications and libraries reduced the use of static salts. They also improved on the number of iterations used to derive keys for PBE. Interestingly, we found that libraries used by the top applications (the T15 dataset) were significantly better at using PBKDF. Furthermore, we found that, since 2012, both applications and libraries had improved on the use of SecureRandom class. To our surprise, while libraries in the top applications significantly outperformed those in R16, the top applications themselves were significantly worse than those in R16. Considering that the SecureRandom class can seed itself and that re-seeding again does not decrease its entropy, we suggest that this class should always seed itself, even if a seed value is provided to the constructor of the class.

Future research on human factors in security should investigate the impact of warning messages for developers on misuse rates of crypto APIs. In addition, to supplement the warning messages, Google can provide to application developers "ready-to-use" code snippets in Android Studio IDE. This would eliminate the necessity for the developers to search online for code examples that, potentially, might have implementation issues.

[7]http://crypto.stackexchange.com/

While our results suggest that there is a positive trend, research community should focus on how to improve the state of the practice even further. For example, one might consider showing a message to the application developers that describes the implications of using static salt and fewer than 1,000 iterations. Such a warning message might include time estimates of how long a password guessing attack would take to go through the password space.

It also worth mentioning that insecure ciphers, namely DES and RC4, are still among the top three most used ciphers. Even more, the popularity of triple DES has decreased eight folds. While it is unclear why DES and RC4 gained popularity, future research should focus on ways to reduce their usage. For instance, one might consider similar warning messages, when a developer tries to use these ciphers.

9 LIMITATIONS AND FUTURE WORK

By using static analysis we inherited all the limitations that come with it. In particular, our super Control Flow Graph (sCFG) is an over estimation of the actual sCFG. This creates a risk to validity of our results, where we analyze a path that never gets executed. While dynamic analysis might address this issue, it is impractical on large sets of applications. We leave the extension of BinSight with a dynamic analysis for future research.

While one cannot obfuscate calls to platform APIs, such as crypto APIs, it is still possible to hide them. In particular, one can use Java Reflection APIs to side-load a binary that would make the actual call to the APIs under investigation. Since, in this research, we did not study the use of Java Reflection API for hiding crypto APIs calls, future research should consider addressing this limitation.

Even though the ratio of fully obfuscated classes in our datasets was negligible (2.5% in R16), understanding how fully obfuscated applications differ from others is still an important and interesting research question to investigate. Such low adoption of full obfuscation, on the other hand, allowed us to use trivial yet efficient and effective source attribution based on package names.

Finally, while looking into the top libraries we found that not all misuses of crypto APIs necessarily have security implications. Our analysis, however, was exploratory in nature and does not provide precise assessment of what ratio of all identified crypto APIs misuses were functional false positives. Considering that the top library from R16 was responsible for 56% of flagged APK files and it used crypto APIs only for obfuscation, we suspect that a significant portion of misuse cases are such functional false positives. Future research should focus on addressing this knowledge gap.

10 CONCLUSION

We studied how crypto APIs misuse in Android applications has changed between 2012 and 2016. By introducing source attribution to the process, we also were able to examine how misuse rates have changed in applications and libraries separately. Overall, we found that significantly fewer libraries and applications were using ECB mode in 2016. However, libraries have significantly increased the use of static IVs (for ciphers in CBC mode) and static encryption keys. At the same time, applications have significantly reduced the use of static IVs and keys. Both libraries and applications have improved in the use of PBKDFs and SecureRandom, i.e., there was a significant decrease in the use of static salt, fewer than 1,000 iterations and static seed.

We also identified several limitations in the previous research (i.e., the CryptoLint study [20]). In particular, while the authors of CryptoLint did white-list 11 libraries, they missed 249 libraries, which resulted in over-counting, since 70% of identified by them misuse cases originated from 222 libraries. Furthermore, we showed that using APK files ratio with misuses as a measure of crypto APIs misuse is highly biased towards libraries, especially the popular ones. To improve the reporting we suggest to measure also the ratio of call-sites that make a mistake.

Finally, through manual analysis of the top-2 libraries in each dataset, we showed that the current approach used for identification of crypto APIs misuse needs further improvement. In particular, by manually analyzing the top-2 libraries from each dataset we showed that it suffers from a significant rate of functional false positives, i.e., cases when crypto APIs are used for other reasons than confidentiality or integrity protection. We suggest that future research should consider improving the technique of identifying misuse of crypto APIs.

We made BinSight framework available as open source. In addition, we will provide data for the R16 and T15 datasets upon request. For the R12 dataset we refer readers to the authors of CryptoLint.

ACKNOWLEDGMENTS

First of all we want to thank Manuel Egele for sharing dataset from CryptoLint study. Second, we would like to thank Dmitry Samosseiko for his help with the R16 dataset acquisition. Finally, we want to thank all anonymous reviewers for their feedback that helped improving this paper.

REFERENCES

[1] 2015. Android Developers Portal | Encryption | Android Developers. https://source.android.com/devices/tech/security/encryption/index.html. (May 2015). https://source.android.com/devices/tech/security/encryption/index.html

[2] 2015. Apktool - A tool for reverse engineering Android apk files (Version 2.01). http://ibotpeaches.github.io/Apktool/. (July 2015). last accessed June 29, 2015.

[3] 2015. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. (July 2015). last accessed October 29, 2016.

[4] 2015. The Legion of the Bouncy Castle. https://www.bouncycastle.org/. (July 2015). last accessed June 29, 2015.

[5] 2015. Spongy Castle - repackage of Bouncy Castle for Android. https://rtyley.github.io/spongycastle/. (July 2015). last accessed June 29, 2015.

[6] 2015. Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar. (July 2015). last accessed June 29, 2015.

[7] 2016. Android Now Has 1.4 Billion 30-Day Active Users Globally. https://techcrunch.com/2015/09/29/android-now-has-1-4bn-30-day-active-devices-globally/. (2016). Accessed August 2, 2016.

[8] 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS '16)* (2016-10-24).

[9] 2016. Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)). https://github.com/androguard/androguard. (November 2016). last accessed November 16, 2016.

[10] 2017. Direct APK Downloader. Direct APK Downloader. (2017). https://androidappsapk.co/apkdownloader/

[11] 2017. Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 7. http://docs.oracle.com/javase/7/docs/technotes/guides/security/SunProviders.html. (May 2017). last accessed May 15, 2017.

[12] 2018. Cipher (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html. (March 2018). last accessed March 27, 2018.

[13] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2017. Comparing the usability of cryptographic APIs. In *Proceedings of the 38th IEEE Symposium on Security and Privacy.*

[14] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[15] David W Binkley and Keith Brian Gallagher. 1996. Program slicing. *Advances in Computers* 43 (1996), 1–50.

[16] Ivan Cherapau, Ildar Muslukhov, Nalin Asanka, and Konstantin Beznosov. 2015. On the Impact of Touch ID on iPhone Passcodes. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS '15)*. 20.

[17] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[18] Anthony Desnos and Geoffroy Gueguen. 2011. Android: From reversing to decompilation. *Proceedings of Black Hat Abu Dhabi* (2011), 77–101.

[19] Danny Dolev, Cynthia Dwork, and Moni Naor. 1998. Non-malleable cryptography. In *SIAM Journal on Computing*. Citeseer.

[20] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 73–84.

[21] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security.. In *USENIX security symposium*, Vol. 2. 2.

[22] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 50–61.

[23] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. 2001. Weaknesses in the Key Scheduling Algorithm of RC4. In *Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography (SAC '01)*. Springer-Verlag, London, UK, UK, 1–24. http://dl.acm.org/citation.cfm?id=646557.694759

[24] B. Kaliski. 2000. PKCS #5: Password-Based Cryptography Specification Version 2.0. (2000).

[25] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[26] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys '14)*. ACM, New York, NY, USA, Article 7, 7 pages. https://doi.org/10.1145/2637166.2637237

[27] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. Libradar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 653–656.

[28] Ildar Muslukhov, Yazan Boshmaf, Cynthia Kuo, Jonathan Lester, and Konstantin Beznosov. 2013. Know your enemy: the risk of unauthorized access in smartphones by insiders. In *Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services (MobileHCI '13)*. ACM, New York, NY, USA, 271–280. https://doi.org/10.1145/2493190.2493223

[29] A. Popov. 2015. RFC7465 - Prohibiting RC4 Cipher Suites. (Feb 2015). https://tools.ietf.org/html/rfc7465

[30] Rahul Raguram, Andrew M. White, Dibyendusekhar Goswami, Fabian Monrose, and Jan-Michael Frahm. 2011. iSpy: automatic reconstruction of typed input from compromising reflections. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*. ACM, New York, NY, USA, 527–536. https://doi.org/10.1145/2046707.2046769

[31] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. 2014. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *Dependable, Autonomic and Secure Computing (DASC), 2014 IEEE 12th International Conference on*. IEEE, 75–80.

11 APPENDIX

11.1 BinSight Linting Pipeline

The overall analysis pipeline is shown at Figure 4. In what follows, we provide technical details on disassembly, de-duplication, and linting.

Each downloaded APK file undergoes a two-step pre-processing stage before it gets linted. The goal of this stage is to filter out all applications that do not use crypto APIs and remove all duplicate APK files, as described below.

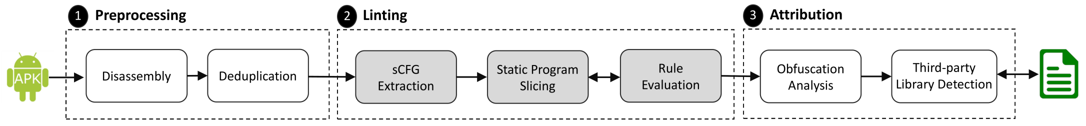

Figure 4: Cryptographic API linting for Android applications using BinSight. Gray components represents parts that were reimplemented from CryptoLint [20], and white components represent the extensions that we added.

11.1.1 Disassembly. Similar to CryptoLint, our analysis operates on a higher-level representation of the Dalvik bytecode. In particular, we use ApkTool [2] to decode an APK file and disassemble it into a set of Smali files. Each Smali file represents a class definition, and uses DEX op-codes to represent instructions [3]. We picked ApkTool over AndroGuard [9], which was used by CryptoLint, to improve analysis reliability. As shown in §7, we were able to analyze all but six applications across the three datasets, while CryptoLint failed to analyze 23% of applications in the original dataset.

After an application is disassembled, we search all its generated Smali files to locate entry points to crypto APIs. If such entry points are not found, the application is disregarded from further analysis. Otherwise, we proceed to the de-duplication step.

11.1.2 De-duplication. Downloading thousands of APK files from Google Play is technically challenging. First, it has to span over weeks or months, in order to avoid account blocking. Second, an application might be listed in multiple categories. These challenges lead to duplicates in a dataset. Removing duplicates is important for validity of the results. For de-duplication we relied on application ID, which is stored in the APK's manifest file.

For each dataset separately we generated a list of all APK filenames, corresponding application Id and its download time (for T15 and R16) or, when available, application version (R12). We then identified all duplicates within a dataset by grouping files with the same application Id. For identified duplicates within a dataset we kept the latest version of the application, based on its download date or version.

11.1.3 Linting. In order to evaluate the rules defined in §3 BinSight computes static program slices that terminate in calls to crypto APIs, and then extracts the necessary information from these slices to evaluate if a corresponding rule was violated or not. We next give a brief overview of the three main steps involved in this stage, and refer the reader to related work for more details [20, 31].

11.1.4 Super Control Flow Graph extraction. It is typical for an application to use crypto APIs in multiple methods. For example, a cipher object could be instantiated in an object constructor and then used in two different methods to encrypt and decrypt the data, respectively. If the two methods are analyzed in isolation, we will not be able to extract the encryption scheme that was used when the cipher object was instantiated. Fortunately, the super control-flow graph (sCFG) of an application allows us to perform inter-procedural analysis, which is required to correlate the use of a cipher object for encryption and decryption with its instantiation.

Endpoint signature	Rule
Cipher.getInstance()	1
cipher.init()	2
secureRandom.setSeed()	6
new SecretKeySpec()	3
new PBEKeySpec()	4
new PBEParameterSpec()	5
new SecureRandom()	6

Table 7: Cryptographic API endpoints and related rules.

BinSight constructs the sCFG of a preprocessed application as follows: First, it extracts the intra-procedural CFGs of all methods from the decoded Smali class files. This task also involves translating all methods into single static assignment (SSA) form [17], and extracting the class hierarchy of all classes in the application. After that, BinSight superimposes a control-flow graph over the CFGs of the individual methods, resulting in the sCFG. Similarly to CryptoLint [20], BinSight adds call edges between call instructions and method entry points, and method exit points are connected with exit edges back to the call site. Similar to CryptoLint, BinSight reconstructs an over-approximated sCFG of the application.

11.1.5 Static program slicing. Static program slicing is the computation of a set of program statements, called slices, that may affect the values of certain variables at a particular program point of interest, referred to as a slicing criterion [15]. BinSight applies static program slicing on the sCFG to identify if the analyzed application uses any of the crypto APIs. In particular, BinSight searches the sCFG for nodes that belong to Java's crypto APIs endpoints. If these nodes are found, it uses their incoming edges to locate all call sites in the application. Note that this search depends on the type of the crypto APIs endpoint in the sCFG. Table 7 shows the relevant API endpoints and their corresponding cryptographic rules.

11.1.6 Rule evaluation. Rule evaluation depends on the values assigned to the parameters of crypto API call, where value assignment can be either local or external to the containing method. For the earlier case, BinSight computes a backward slice of the program to all possible locations where the involved parameter is set, after which we apply validation logic on its value. As for the latter case, the evaluation depends on the origin of value assignment outside the method. As such, BinSight computes backward slices to all locations where this value can be assigned. BinSight stops the computation if it reaches a dead-end, where a node does not have any incoming edge or it reaches an assignment to a static value.

11.2 Rule-based classification

The following rules were defined as a result of several manual iterations over all unique class identifiers. We ran these rules several times over all unique class identifiers and every time manually analyzed the results. Our main objective at this point is to find patters that we can include into our classifier. Eventually we came up with the seven rules (listed below) that allowed us assign automatically the level of class identifier renaming (CIR), i.e., none, class, partial, and full CIR obfuscation.

(1) If all parts of the identifier are of length one, then this case is full obfuscation.
(2) If all but the first part of the identifier are of length one and the first part is in the set {com, ch, org, io, jp, net}, then this case is partial obfuscation.
(3) If none of the package name parts in the identifier are of length one, then this case is either none or class-level obfuscation.
(4) if at least one part but not all of the identifier are of length one, then the case is partial obfuscation.
(5) If class name is longer than 3 chars then it is none obfuscation.
(6) If class name length is 1 character, then this case is class obfuscation.
(7) If class name of length 2 or 3 characters and the first character is in lower case, then this is class obfuscation.