Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild

Shuaike Dong[1], Menghao Li[2], Wenrui Diao[3], Xiangyu Liu[4], Jian Liu[2(✉)], Zhou Li[5], Fenghao Xu[1], Kai Chen[2], XiaoFeng Wang[6], and Kehuan Zhang[1(✉)]

[1] The Chinese University of Hong Kong, Sha Tin, Hong Kong
{ds016,xf016,khzhang}@ie.cuhk.edu.hk
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{limenghao,liujian6,chenkai}@iie.ac.cn
[3] Jinan University, Guangzhou, China
diaowenrui@link.cuhk.edu.hk
[4] Alibaba Inc., Hangzhou, China
eason.lxy@alibaba-inc.com
[5] ACM Member, Boston, MA, USA
lzcarl@gmail.com
[6] Indiana University Bloomington, Bloomington, IN, USA
xw7@indiana.edu

Abstract. Program code is a valuable asset to its owner. Due to the easy-to-reverse nature of Java, code protection for Android apps is of particular importance. To this end, code obfuscation is widely utilized by both legitimate app developers and malware authors, which complicates the representation of source code or machine code in order to hinder the manual investigation and code analysis. Despite many previous studies focusing on the obfuscation techniques, however, our knowledge of how obfuscation is applied by real-world developers is still limited.

In this paper, we seek to better understand Android obfuscation and depict a holistic view of the usage of obfuscation through a large-scale investigation in the wild. In particular, we focus on three popular obfuscation approaches: identifier renaming, string encryption and Java reflection. To obtain the meaningful statistical results, we designed efficient and lightweight detection models for each obfuscation technique and applied them to our massive APK datasets (collected from Google Play, multiple third-party markets, and malware databases). We have learned several interesting facts from the result. For example, more apps on third-party markets than malware use identifier renaming, and malware authors use string encryption more frequently. We are also interested in the explanation of each finding. Therefore we carry out in-depth code analysis on some Android apps after sampling. We believe our study will help developers select the most suitable obfuscation approach, and in the meantime help researchers improve code analysis systems in the right direction.

Keywords: Android · Obfuscation · Static analysis · Code protection

R. Beyah et al. (Eds.): SecureComm 2018, LNICST 254, pp. 172–192, 2018.
https://doi.org/10.1007/978-3-030-01701-9_10

1 Introduction

Code is a very important intellectual property to its developers, no matter if they work as individuals or for a large corporation. To protect this property, *obfuscation* is frequently used by developers, which is also considered as a double-edged sword by the security community. To a legitimate software company, obfuscation keeps its competitors away from copying the code and quickly building their own products in an unfair way. To a malware author, obfuscation raises the bar for automated code analysis and manual investigation, two approaches adopted by nearly every security company. For a mobile app, especially the one targeting Android platform, obfuscation is particularly useful, given that the task of disassembling or decompiling Android app is substantially easier than doing so for other sorts of binary code, like X86 executables.

Android obfuscation is pervasive. On the one hand, there are already more than 3.5 million apps available for downloading just in one app market, Google Play, up to December 2017 [13]. On the other hand, many off-the-shelf obfuscators are developed, like ProGuard [14], DashO [7], DexGuard [8], DexProtector [9], etc. Consequently, the issues around app obfuscation attract many researchers. So far, most of the studies focus on the topics like what obfuscation techniques can be used [20], how they can be improved [38], how well they can be handled by state-of-art code analysis tools [37], and how to deobfuscate the code automatically [22]. While these studies provide solid ground for understanding the obfuscation *techniques* and its *implications*, there is still an unfilled gap in this domain: how is obfuscation *actually used* by the vast amount of developers?

We believe this topic needs to be studied, and the answer could enlighten new research opportunities. To name a few, for developers, learning which obfuscation techniques should be used is quite important. Not all obfuscation techniques are equally effective, and some might even have bad influence on the performance of a program. Plenty of code analysis approaches were proposed, but their effects are usually hampered by obfuscation and the impact greatly differs based on the specific obfuscation technique in use, e.g., identifier renaming is much less of an issue comparing to string encryption. Knowing the preferences of obfuscation techniques can better assist the design of code analysis tools and prioritize the challenges need to be tackled. All roads paving to the correct conclusions call for measurement on real-world apps, and only the result coming from a comprehensive study covering a diverse portfolio of apps (published in different markets, in different countries, from both malware authors and legitimate companies) is meaningful.

Our Work. As the first step, in this paper, we systematically study the obfuscation techniques used in Android apps and carry out a large-scale investigation for apps in the wild. We focus on three most popular Android obfuscation techniques (identifier renaming, string encryption, and Java reflection) and measure the base and popular implementation of each technique. To notice, the existing tools, like deobfuscators, cannot solve our problem here, since they either work well against a specific technique or a specific off-the-shelf obfuscator (e.g.,

ProGuard). As such, they cannot be used to provide a holistic view. Our key insight into this end is that instead of mapping the obfuscated code to its original version, a challenge not yet fully addressed, we only need to *cluster* them based on their code patterns or statistical features. Therefore, we built a set of lightweight detectors for all studied techniques, based on machine learning and signature matching. Our tools are quite effective and efficient, suggested by the validation result on ground-truth datasets. We then applied them on a real-world APK dataset with 114,560 apps coming from three different sources, including Google Play set, third-party markets set, and malware set, for the large-scale study.

Discoveries. Our study reveals several interesting facts, with some confirming people's intuition but some contradicting to common beliefs: for example, as an obfuscation approach, identifier renaming is more widely-used in third-party apps than in malware. Also, though basic obfuscation is prevalently applied in benign apps, the utilization rate of other advanced obfuscation techniques is much lower than that of malware. The detailed statistical results are provided in Sect. 4. We believe these insights coming from "*big code*" are valuable in guiding developers and researchers in building, counteracting or using obfuscation techniques.

Contributions. We summarize this paper's contributions as below:

- **Systematic Study.** We systematically study the current mainstream Android obfuscation techniques used by app developers.
- **New Techniques.** We propose several techniques for detecting different obfuscation techniques accurately, such as n-gram -based renaming detection model and backward slicing-based reflection detection algorithm.
- **Large-scale Evaluation.** We carried out large-scale experiments and applied our detection techniques on over 100K APK files collected from three different sources. We listed our findings and provided explanations based on in-depth analysis of obfuscated code.

Roadmap. The rest of this paper is organized as follows: We systematically summarize popular Android obfuscation techniques in Sect. 2.2. Section 3 overviews the high-level architecture of our detection framework. The detailed detection strategies and statistical results on large-scale datasets are provided in Sect. 4. Also, we discuss some limitations and future plans in Sect. 5. Section 6 reviews the previous research on Android obfuscation, and Sect. 7 concludes this paper.

2 Background

In this section, we briefly introduce the structure of APK file and overview some common Android obfuscation techniques.

2.1 APK File Structure

An APK (Android application package) file is a zip compressed file containing all the content of an Android app, in general, including four directories (`res`, `assets`, `lib`, and `META-INF`) and three files (`AndroidManifest.xml`, `classes.dex`, and `resources.arsc`). The purposes of these directories and files are listed as below.

`res` This directory stores Android resource files which will be mapped to the `.R` file in Android and allocated the corresponding ID.

`assets` This directory is similar to the `res` directory and used to store static files in the APK. However, unlike `res` directory, developers can create subdirectories in any depth with the arbitrary file structure.

`lib` The code compiled for specific platforms (usually library files, like `.so`) are stored in this directory. Subdirectories can be created according to the type of processors, like armeabi, armeabi-v7a, x86, x86_64, mips.

`META-INF` This directory is responsible for saving the signature information of a specific app, which is used to validate the integrity of an APK file.

`AndroidManifest.xml` This XML file is the configuration of an APK, declaring its basic information, like name, version, required permissions and components. Each APK has an AndroidManifest file, and the only one.

`classes.dex` The dex file contains all the information of the classes in an app. The data is organized in a way the Dalvik virtual machine can understand and execute.

`resources.arsc` This file is used to record the relationship between the resource files and related resource ID and can be leveraged to locate specific resources.

2.2 Android Obfuscation Characterization

In general, obfuscation attempts to garble a program and makes the source or machine code more difficult for humans to understand. Programmers can deliberately obfuscate code to conceal its purpose or logic, in order to prevent tampering, deter reverse engineering, or behave like a puzzle for someone reading the code. Specifically, there are several common obfuscation techniques used by Android apps, including identifier renaming, string encryption, excessive overloading, and so forth.

Identifier Renaming. In software development, for good readability, code identifiers' names are usually meaningful, though developers may follow different naming rules (like CamelCase, Hungarian Notation). However, these meaningful names also accommodate reverse-engineers to understand the code logic and locate the target functions rapidly. Therefore, to reduce the potential information leakage, identifier's names could be replaced by meaningless strings. In the following example, all identifiers in class `Account` are renamed.

```
1 public class a{
2     private Integer a;
```

```
    private Float = b;
    public void a(Integer a, Float b){
        this.a = a + Integer.valueOf(b)
    }
}
```

String Encryption. Strings are very common-used data structures in software development. In an obfuscated app, strings could be encrypted to prevent information leakage. Based on cryptographic functions, the original plaintexts are replaced by random strings and restore at runtime. As a result, string encryption could effectively hinder *hard-coded* static scanning. The following code block shows an example.

```
String option = "@^@#\x ‘1 m*7 %**9_!v";
this.execute(decrypt(option));
```

Java Reflection. Reflection is an advanced feature of Java, which provides developers with a flexible approach to interact with the program, e.g., creating new object instances and invoking methods dynamically. One common usage is to invoke nonpublic APIs in the SDK (with the annotation `@hide`). The following code gives an example of reflection that invokes a hidden API `batteryinfo`.

```
Object object = new Object();
Method getService = Class.forName("android.os.
    ServiceManager").getMethod("getService", String.class);
Object obj = getService.invoke(object, new Object[]{new
    String("batteryinfo")});
```

As an obfuscation technique, reflection is a good choice of hiding program behaviors because it can transfer the control to a certain function implicitly, which can not be well handled by state-of-the-art static analysis tools. Therefore, malware developers usually heavily employ reflection to hide malicious actions.

3 System Design

Our target is to systematically study the Android obfuscation techniques and carry out a large-scale investigation. As the first step, we design an efficient Android code analysis framework to identify the obfuscation techniques used by developers. Here we overview the high-level design of this framework and introduce the datasets prepared for the subsequent large-scale investigations.

3.1 System Overview

To detect the usage of obfuscation techniques, we propose an architecture to analyze APK files automatically, as illustrated in Fig. 1. After the APK files collected from several channels (details are provided in Sect. 3.2) are stored on

our server, this detection framework will try to unpack them for the primary testing. Some damaged APK files failing to pass this step will be discarded and manually checked to make sure the samples used in the following phases are valid. Then this framework applies three targeted detection methods to identify obfuscated Smali code blocks. These detection methods could be classified into two categories: signature-based and machine learning-based.

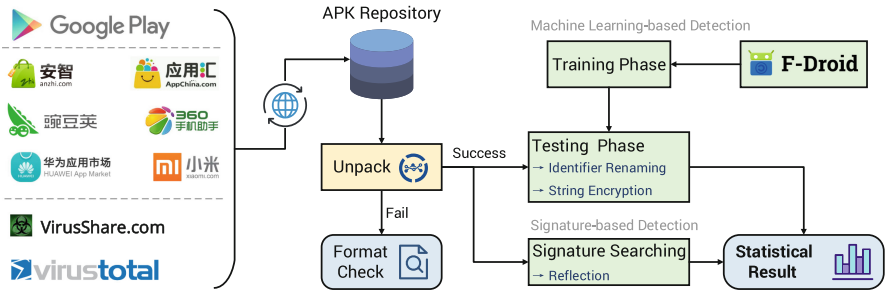

Fig. 1. Android app obfuscation detection framework

3.2 APK Dataset

We are interested in the obfuscation usage status of apps in different types, so three representative APK datasets were used in our experiment: Google Play set (26,614 samples), third-party market set (65,666 samples), and malware set (22,280 samples). These samples were collected during 2016 and 2017. In total, our experiment dataset contains 114,560 sample with the size of around 1.521TB. More details are given in Table 1.

As the official app store for Android, Google Play is the main Android app distribution channel. Thus, its sample set could reflect the deployment status of obfuscation used by mainstream developers. Also, due to the policy restriction, in some countries (such as China), Google Play is not available, and users have to install apps from third-party markets. Therefore, in the second dataset, we select six popular app markets from China (say Anzhi [4], Xiaomi [19], Wandoujia [18], 360 [1], Huawei [10], and AppChina [5]) and developed the corresponding crawlers to collect their apps. Note that the replicated samples from different markets have been excluded. Lastly, except for legitimate app samples, we are also curious about whether malware authors heavily use obfuscation skills to hide their malicious intentions. So, the last dataset contains the malware samples coming from VirusShare [16] and VirusTotal [17,30].

4 Obfuscation Detection and Large-Scale Investigation

In this section, we introduce the detection approaches for each obfuscation technique and summarize our findings based on large-scale experiments.

Table 1. APK dataset for investigation

Type	Source	Number
Official Market	Google Play	26,614
3rd-party Market	Wandoujia	8,979
	360	18,724
	Huawei	22,048
	Anzhi	7,121
	Xiaomi	4,649
	AppChina	4,145
Malware	VirusShare	19,004
	VirusTotal	3,267

4.1 Identifier Renaming

Generally, in the software development, the names of identifiers (variable names, function names, and so forth) are usually meaningful, which could provide good code readability and maintainability. However, such *clear* names may leak much information due to the easy-to-reverse feature of Java. As a solution, identifier renaming is proposed and widely used in practice.

The renaming operation can be appended at different stages of APK file packaging. For example, ProGuard [14] and Allatori [2] work at the source-code level, mapping the original names to mangled ones based on the user's configuration. The other obfuscators, like DashO [7], DexProtector [9], and Shield4J [15], can work directly on APK files, modifying `.class` and `.dex` files.

Given an identifier, we can easily tell whether some obfuscator has renamed it based on the information it contains. In other words, if an identifier name is obscure and meaningless, it can be regarded as obfuscated because it tries to hide the actual intention. A typical renaming operation is changing the original name to a single character (like "a", "b") or some kind of puzzling string (like "lllllll", "oO00O0oo") [20]. However, the manual check is obviously not qualified for our large-scale scanning goal. Moreover, we focus on the whole APK contents rather than a single identifier. Therefore, we need to design a representation which can measure the overall extent of identifier renaming given an app.

Beyond that, as a special case of identifier renaming, the *excessive overloading* technique utilizes the overloading feature of Java and could map irrelevant identifier names to the same one, making the code more confusing to analysts [21]. For example, in the sample `idfhn`[1], more than 46 functions are named as `idfhn` (the same as the package name). Though the compiler could distinguish these variables with the same name, security analysts have to face more troubles. In our research, we also paid attention to the application of overloading feature and its impact on code analysis.

[1] MD5: `7d9eb791c09b9998336ef00bf6d43387`.

Identifier Renaming Detection. To the above challenges and targets, we combine the computational linguistics and machine learning techniques for accurate renaming detection. The high-level idea is based on the probabilistic language model. The insight is that identifier renaming will lead to the abnormal distribution of characters and character combinations, which distinguishes from normal ones (non-obfuscated). The model outputs 1 or 0 according to whether the app is judged as using identifier renaming. Here we give our three-step approach:

1. *Data Pre-processing.* All the identifier names of the target APK sample are extracted as the training candidates. Note that, software developers often introduce third-party libraries into their apps. However, those third-party libraries may contain obfuscated code, which does not reflect the protection deployed by developers. Therefore we also pre-removed over 12,000 common third-party libraries to avoid the inference using the approach of Li et al. [32].
2. *Feature Generation.* The amount of identifiers varies among different apps. To build a uniform expression, we apply the n-gram algorithm [12] to generate a fixed-length [2] feature vector for each app. An n-gram is a contiguous sequence of n items from a given sequence of text or speech. Through our small-scale tests, we found 3-gram [3] can well depict the distribution of character combinations while restricting the length of the vector. Then we applied it to traverse each name string in extracted raw name set to form the feature vector. Each element of the vector records the frequency of a certain character combination and will be normalized. Note that, the vector also involves the frequencies of fewer-than-three character combinations (`a`, `ab`, etc.) due to the length of an identifier may be smaller than three.
3. *Classification.* We collected apps from F-Droid and applied different obfuscators on them to generate the training set. Due to our model is a two-class classifier, we decided to use Supported Vector Machine (SVM) as the classification algorithm for its powerful learning ability. After the training phase, we applied it to our large-scale dataset.

Experiment Settings. We implemented a prototype of our detection model based on Androguard [3] with more than 1,500 Python lines of code. For training, we downloaded 3,147 apps with their corresponding source code from F-Droid. Two obfuscators, ProGuard and DashO, were used to generate variant obfuscated samples because they have different renaming policies. Note that, due to the diversity of apps' project configurations, not all of them can be processed by both ProGuard (2,107 successful samples) and DashO (654 successful samples). Among them, we randomly chose 500 original apps and 500 successful obfuscated apps (250 from Proguard and 250 from DashO) as the training set. We

[2] The length is restricted by the legal characters sets used for contracting a name in Java: [“a-z”, “ A-Z”, “0-9”, “_”, “$”, “\”].

[3] For example, if there is a string “abcdefgh”, all of the 3-gram sequences it contains are {abc, bcd, cde, def, efg, fgh}.

then randomly selected 500 original, 250 Proguard-obfuscated and 250 DashO-obfuscated apps from the remaining set to do the validation. Our model reached 0.6% FN rate and 0.0% FP rate, which is quite satisfactory. We then collected another testing set consisting of 200 samples obfuscated by another obfuscator called Allatori. The completely successful classification results showed the strong generalization ability of our model.

Large-Scale Investigation and Findings. We carried out a large-scale detection on the three typical datasets (Google Play, third-party markets, and malware) mentioned in Sect. 3.2. The obfuscation detection result by dataset is given in Fig. 2. According to such statistics, we have two immediate findings:

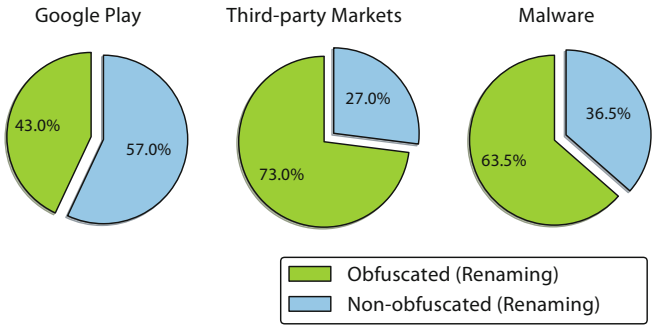

Fig. 2. Ratio of identifier renaming in three datasets

⇒ 1. Compared with the apps on Google Play, the ones from third-party markets apply more renaming operations.
⇒ 2. Over one third of malware don't apply identifier renaming.

To the first finding, we ascribe it to the discrepancy between app market environments. The piracy issue in Chinese app markets are quite severe, say nearly 20% apps are repacked or cloned [24]. Such situation urges developers to put more effort into protecting their apps. On the other hand, Google Play provides more strict and timely supervision, which mitigates the severity of software piracy largely. The better application ecosystem makes many developers believe obfuscation is just an optional protection approach.

To the second finding, the percentage of malware utilizing identifier renaming is only 63.5%, slightly less than third-party apps, which is opposite our traditional opinion. After manually checking the code of malware without renaming-obfuscation, we conclude that two aspects contribute to such phenomenon.

– *Script Kiddies.* Many entry-level malware authors only could develop simple malicious apps and lack the knowledge of how to disguise malicious behaviors through obfuscation. A few codes and clumsy class structures are two main

features of those entry-level apps. The vicious behaviors of the malware are usually exposed to analysts due to the rough implementation.
– *False Alarmed "Malware"*. For some apps, their main bodies are benign and non-obfuscated, while the imported third-party libraries contain some kinds of sensitive and suspicious behaviors which are recognized as malicious by some anti-virus software. A common example is the advertising library.

In addition, we explored the difference in renaming implementation between malware and benign apps. The result reflects:

⇒ 1. Malware authors prefer to use more complex renaming policies.
⇒ 2. Malware may use irrelevant names to hide the true intention.

We find that, in benign apps (the samples on Google Play and third-party markets), most identifier names are mapped to {a, b, aa, ab, aaa, ...} and so on, in lexicographic order. In fact, such renaming rules accord with the default configurations of many obfuscators (such as ProGuard). That is to say, app developers do not intend to change the renaming rules to more ingenious ones. However, malware authors usually put more effort into configuring the renaming policies. For example, some malware samples utilize special characters (encoded in Unicode) as obfuscated names (e.g., È, ô), which seems very odd but still be regarded as legal by Java compilers. Also, some dazzling weird names (like {lllllllll, oO00O0oo, ...}) could be found. Such renaming policy can actually make manual analysis more strenuous.

Apart from that, we find that overloading, as a grammar feature provided by Java, is also applied by malware to confuse analysts. In sample `tw.org.ncsist. mdm`[4], the name of overloaded function `attachBaseContext` (A `protected` method in class `android.app.Application`) will mislead security analysts because the logic of this function is actually implemented for encryption.

4.2 String Encryption

The strings in a `.dex` (Dalvik executable) file may leak a lot of private information about the program. As security protection, those hard-coded texts can be stored in an encrypted form to prevent reverse analysis. In this section, we take a deep insight into the string encryption and focus on two aspects:

1. Detect whether an app uses the string encryption.
2. Analyze the cryptographic functions invoked by apps.

String Encryption Detection. Similar to the approach for identifier renaming detection (Sect. 4.1), we trained a machine-learning based model to classify encrypted strings and plain-text strings. We reused the 3-gram algorithm, SVM algorithm, and the open-source apps from F-Droid. Here we only describe the different steps. At first, all strings appeared in an app are extracted. Next, a

[4] MD5: `01a93f7e94531e067310c1ee0f083c07`.

vector was generated for each app. Distinct from the setting for identifier renaming detection, there is no restriction on the content of a string. Therefore, we extended the acceptable character set to all ASCII codes[5].

In the implementation, we reused most code of identifier renaming detection model. Since string encryption is not a common function provided by off-the-shelf obfuscators, we chose DashO and DexProtector to generate the ground truth and finally obtained 737 string-encrypted samples for training. To avoid the overfitting caused by unbalanced data, we randomly selected 500 original apps and 500 string-encrypted apps to train our model. To verify the effectiveness, we randomly selected another 100 original apps and 100 string-encrypted apps for testing. The result shows our model could achieve 98.5% success rate with FP 1% and FN 2%.

Cryptographic Function Detection. Previous work has proposed various approaches to identify cryptographic functions in a program, like [23,28,34]. Those methods were specifically designed for the identification of the standard, modern cryptographic algorithms in binary code, like AES, DES, and RC4. The features used by the previous commonly include entropy analysis, searchable constant patterns, excessive use of bitwise arithmetic operations, memory fetch patterns and so on, besides, the dynamic binary instrument is also widely-used by analysts to better locate and identify the cryptographic primitives. However, previous approaches do not fit android platform very well due to three reasons: (1) Smali instructions have different representations from the x86 assembly language, especially for memory access. (2) Java provides the complete implementations of standard cryptographic algorithms through Java Cryptography Extension [11]. Therefore, in most cases, developers do not need to implement cryptographic related functions again. (3) Java provides a series of string & character operations, like `concat()`, `substring()`, `getChars()`, `strim()` and so on, which can be used to build an encrypted string.

To better handle the identification in Android apps, we extended the previous approaches with more empirical features, shown as below.

- The ratio of bit and loop operations.
- The usage of Java Cryptography Extension API invoking.
- The amount of operations on string & character variables.
- The frequency of encrypted strings as function parameters (for decryption).

Large-Scale Investigation and Findings. We applied our string encryption detection model on the testing datasets. The results are presented in Fig. 3. The direct findings are that:

⇒ 1. Nearly all benign apps don't use string encryption.
⇒ 2. String encryption is more popular in malware.

[5] Unicode codes can be represented in the form of $\backslash uxxxx$, where $xxxx$ is a 4-digit hexadecimal number.

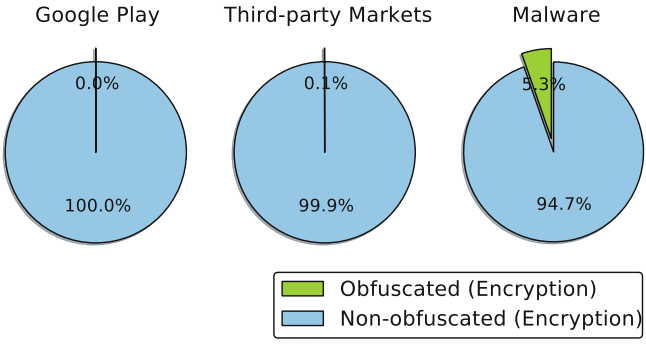

Fig. 3. Ratio of string encryption in three datasets

These statistical results comply with our perception, and we could understand it from three perspectives. (1) String encryption is not a common feature provided by off-the-shelf obfuscators (Proguard). The obfuscators offering the string encryption feature are expensive (DexGuard, DexProtector). (2) Many developers may lack the knowledge or awareness of deploying more advanced obfuscation techniques. They may believe the default identifier renaming is enough for code protection and it is not necessary to consider other techniques. (3) String encryption can help malware evade the signature scanning of some anti-virus software and hidden the intention effectively, leading to a higher rate of utilization than benign apps.

We then manually analyzed the implementations of cryptographic functions extracted from malware set and got the following findings.

⇒ The cryptographic functions usually disguise its true intention by changing to an irrelevant name.

For instance, in sample `com.solodroid.materialwallpaper`[6], the decryption function is disguised as a common legitimate API `NavigationItem;->getDrawable()` which should be used for retrieving a drawable object.

⇒ About 17.6% of string-encrypted malware implement multiple cryptographic functions and take turns to use them in a single app.

In sample `com.yandex.metrica`[7], four different cryptographic functions were implemented. All of them first initialize the key, then doing the encryption/decryption. However, the key initialization procedures are quite different from each other. As a result, the workload of restoring rises significantly for analysts.

```
1 // In class com.yandex.metrica.impl.ad;
2 static final String a(String str){
3 if (c == null){
```

[6] MD5: `fab2711b0b55eb980f44bfebc2c17f1f`.
[7] MD5: `95f7d37a60ef6d83ae7443a3893bb246`.

```
    a13840(); // key initialization function
}
    Continue ...
}
```

⇒ The secret keys can be either statically defined or dynamically generated.

In the static case, the key is either hard-coded or directly imported as the parameter, which can be easily located and obtained. On the other hand, the dynamic key is usually generated at runtime and even could be fluctuating in different runtime context, which is nearly impossible to be handled by static analysis. The following code snippet shows an example of dynamic key generation, in which `elements[3]` is not a fixed value because of the uncertain stack trace at runtime.

```
StackTraceElement[] elements = Thread.currentThread().
    getStackTrace();
int hashCode = elements[3].getClassName()+elements[3].
    getMethodName().hashCode();
```

4.3 Reflection

Reflection allows programs to create, modify and access an object at runtime, which brings many flexibilities. However, such dynamic feature also impedes static analysis due to those reflective invocations, especially those invoking other functions. Such uncertain behaviors could result in that the static analysis cannot capture the real intention.

In this section, we explore two questions on reflection:

1. How widespread is the reflection used in the wild?
2. Among all the usage, how many of them are for the obfuscation purpose?

Reflection provides diverse APIs targeting at different objects like Class, Method and Field. In practice, particular APIs are often executed in sequence to achieve specific functionalities. In our study, we focus on the sequence pattern [`Class.forName()` → `getMethod()` → `invoke()`] which is the most frequent pattern for reflective calls mentioned by Li et al. [31]. Also, in this sequence, the execution of the program is implicitly transferred to another function (the function targeted by `getMethod()`), which has an obvious influence on program status, especially the control flow.

Reflection Detection. First, we located the reflective invocations by searching for the certain APIs, `Class.forName()`, etc. Then we managed to recover the real target of the reflective calls, actually the parameters of `Class.forName()` and `getMethod()`. In theory, dynamic analysis is the best way to find the input parameter. However, its low path coverage and efficiency issues are not suitable for large-scale scanning. To balance the efficiency and coverage, we developed a

light-weight tool to trace the input parameters. The high-level idea is to find the real content of the parameters through backward slicing.

More details, first our tool scans the function body and locates two reflection calls – `Class.forName()` and `getMethod()`. The parameter registers will be set as *slicing criterion*. Then it traces back from the locations, analyzing each instruction to find the corresponding *slices*. After that, this tool parses and simulates each instruction in *slices*, and calculates the final value of the *slicing criterion*.

Here, we use a real-world example (see the below code block) to illustrate such work flow. In this case, our tool will mark the positions of *blue-highlighted* reflective calls and trace the data flow of *red-highlighted* registers. The final output would be {"`android.os.SystemProperties`", "`get`"}.

```
1 const/4 v1, 0
2 const-string v0,'android.os.SystemProperties'
3 invoke-static v0,Ljava/lang/Class;->forName(Ljava/lang/String
      ;)Ljava/lang/Class;
4 const-string v2, 'get'
5 ...
6 invoke-virtual v0, v2, v3, Ljava/lang/Class;->getMethod(Ljava/
      lang/String; [Ljava/lang/Class;)Ljava/lang/reflect/
      Method;
```

Note that, to reduce the maintenance complexity, we do not carry out recursive function invoking resolution. If the content of the target register is the return value of another function, the metadata of that function will be recorded (name, parameters, etc.). Besides, due to our tool works at the static level, predicates (`if` and `switch`, etc.) may lead to the failure of recovering the real target. Whenever the target can not be definitely obtained by our tool, a `null` will be recorded instead. We then measured the successful recovery rate of our static-level tool. Among all 121,262 occurrences of reflective calls, 116,595 (96.2%) `non-null` targets were recorded, which means our tool can work effectively.

Large-Scale Investigation and Findings. The implementation of our detection model (reflection usage and invoked functions in reflection) is still based on Androguard with around 1600 Python lines of code. After experiments on our APK dataset, the reflection statistics are shown in Fig. 4. We could find:

⇒ The proportions of reflection deployment in benign apps and malware are similar.

To the successfully recovered functions, we further explore why these reflection implementations are necessary. According to different APK dataset, the most frequently invoked functions are listed in Tables 2, 3, and 4 respectively. These lists reflect:

⇒ Most of the reflection cases are used to invoke hidden functions or to support backward compatibility.

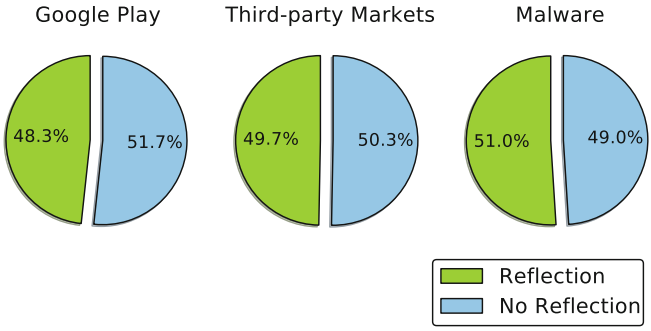

Fig. 4. Ratio of reflection in three datasets

Table 2. Functions invoked via reflection (Google Play)

Frequency	Recovered Function
2,275	`android.support.v4.content.LocalBroadcastManager.getInstance`
1,297	`android.webkit.WebView.onPause`
1,250	`android.os.SystemProperties.get`
821	`org.apache.harmony.xnet.provider.jsse.NativeCrypto.RAND_seed`
523	`com.google.android.gms.common.GooglePlayServicesUtil.isGooglePlayServicesAvailable`

Table 3. Functions invoked via reflection (3rd-p Market)

Frequency	Recovered Function
3,859	`android.os.SystemProperties.get`
1,800	`android.support.v4.content.LocalBroadcastManager.getInstance`
1,158	`org.apache.harmony.xnet.provider.jsse.NativeCrypto.RAND_seed`
721	`android.os.ServiceManager.getService`
613	`android.os.Build.hasSmartBar`

Table 4. Functions invoked via reflection (malware)

Frequency	Recovered Function
2,977	`java.lang.String.valueOf`
2,142	`android.telephony.gsm.SmsManager.getDefault`
687	`android.os.SystemProperties.get`
518	`java.lang.String.charAt`
352	`java.lang.String.equals`

In Android system, the functions related to the Android framework and OS itself are usually annotated with the label "`@hide`", which can only be called through reflection. In above three tables, all functions starting with `android.os.*` and `android.webkit.*` are hidden-annotated.

We also manually checked the use case of `android.support.v4.content.LocalBroadcastManager.getInstance`. We found that the corresponding reflective calls are usually enclosed in a *try-catch* block, aiming to handle the not-found exception caused by discrepancy among systems with different versions. Such pattern is a programming standard recommended by the official Android documents [6].

To malware samples, we find:

⇒ Compared with benign apps, malware prefers to use more complex reflection invoking patterns to hide its intentions.
⇒ String operations are usually combined with reflection to enhance the complexity of the code.

For example, the following code block is extracted from an obfuscated malware[8]. After analysis, the function invoked by reflection could be restored as:

```
1 if (!ò.trim().toLowerCase().contains(Ô("G))OCH")))
```

As a comparison, the original code is shown below. In this case, all string operations can be written in non-reflection forms. We could find such reflection usage makes the code structure more complicated and confusing, which enhances the effect of code obfuscation.

```
1 if (!((Boolean) Class.forName("java.lang.String").
    getMethod("contains", new Class({CharSequence.class}).
    invoke(Class.forName("java.lang.String").getMethod("
    toLowerCase", null).invoke(Class.forName("java.lang.
    String").getMethod("trim", null).invoke(ò, null), null)
    , new Object[]{Ô("G))OCH")})).booleanValue())
```

5 Discussion

In this section, we discuss some limitations of our study and then describe the future plan. Though we have conducted a large-scale investigation of mainstream obfuscation techniques used in Android apps, we should point out there are still some existing techniques not involved in our research, say control flow obfuscation and native code obfuscation.

According to our investigation, the control flow obfuscation is non-universal and only provided by two available Android obfuscators, DashO and Allatori. Moreover, we believe both tools cannot provide a strong control flow obfuscation implementation as they claimed. In our experiments, less than 5% methods

[8] MD5: `7ff1b8afd22c1ed77ed70bfc04635315`.

contained in our sample APKs were obfuscated in the control flow, and the obfuscation implementations were trivial (such as only adding some simple "try-catch" combinations). Therefore, at this moment, we cannot capture enough meaningful (real-world) control-flow obfuscated samples for study.

Another topic not involved in this paper is native code obfuscation. As an advanced programming skill, developers can implement components in native code with the help of Android NDK. However, the implementation of native code is quite different from Java-level techniques, which makes the native code obfuscation could be treated as an independent research topic. Therefore, we leave it as our future study.

6 Related Work

Obfuscation is always a hot research topic in Android ecosystem, and there are several studies performed on how to obfuscate Android apps effectively and how to measure the obfuscation effectiveness.

6.1 Obfuscation Measurement and Assessment

Obfuscation techniques have been widely used in the Android app development. Naturally, in academia, researchers are interested in whether these techniques do work. An early attempt is [27] which empirically evaluates a set of 7 obfuscation methods on 240 APKs. Also, Park et al. [35] empirically analyzed the effects of code obfuscation on Android app similarity analysis. Recently, Faruki et al. [26] conducted a survey to review the mainstream Android code obfuscation and protection techniques. However, they concentrated on the technical analysis to evaluate different techniques, not like our work based on a large-scale dataset. They show that many obfuscation methods are idempotent or monotonous. Wang et al. [41] defined the obfuscator identification problem for Android and proposed a solution based on machine learning techniques. The experiments indicated that their approach could achieve about 97% accuracy to identify ProGuard, Allatori, DashO, Legu, and Bangcle. Duan et al. [25] conducted a comprehensive study on 6 major commercial packers and a large set of samples to understand Android (un)packers. On the aspect of deobfuscation research, Bichsel et al. [22] proposed a structured prediction approach for performing probabilistic layout deobfuscation of Android APKs and implemented a scalable probabilistic system called DeGuard.

Different from above research, our work is based on large Android app datasets which cover official Google play store, third-party Android markets, and update-to-date malware families. We attempt to understand the distribution of Android obfuscation techniques and provide the up-to-date knowledge about app protection.

6.2 Security Impact of Android Obfuscation

As discussed earlier, the obfuscation will create barriers for Android program analysis. Works on clone/repackage detection [40,42] find that obfuscations can impair detection results.

Studies of malware detection also showed that obfuscation is an obstacle to malware analysis. Rastogi et al. [37] evaluated several commercial mobile anti-malware products for Android and tested how resistant they are against various common obfuscation techniques. Their experiment result showed anti-malware tools make little effort to provide transformation-resilient detection (in the year 2013). After that, Maiorca et al. [33] conducted a large-scale experiment in which the detection performance of anti-malware solutions are tested against malware samples under different obfuscation strategies. Their results showed the improvement of anti-malware engines in recent years. Recently, Hoffmann et al. [29] developed a framework for automated obfuscation, which implemented fine-grained obfuscation strategies and could be used as test benches for evaluating analysis tools. Similar works are also completed by Preda et al. [36]. To handle obfuscated samples, Suarez-Tangil et al. [39] propose DroidSieve, an Android malware classifier based on static analysis and deep inspection that is resilient to obfuscation.

For malware detection, researchers mainly discussed arms race between obfuscation and malware detection. Although some malware detection tools claim to still work well in the presence of obfuscation, none could eliminate the obfuscation effects in their experimental evaluation. Our study focuses on the empirical study of security impacts of obfuscation in the wild from different views, which are complementary to existing works. That is, we statistically evaluate the distribution of obfuscation methods from views of different markets, hardening capability of obfuscations and temporal evolution, with a light-weight and scalable obfuscation detection framework. We believe some of our findings would be useful for developers and researchers to better understand the usage of obfuscation, for example, keeping pace with the development of obfuscation technique.

7 Conclusion

In this paper, we concentrate on exploring the current deployment status of Android code obfuscation in the wild. For this target, we developed specific detection tools for three common obfuscation techniques and performed a large-scale scanning on three representative APK datasets. The results show that, to different techniques and app categories, the status of code obfuscation differs in many aspects. For example, the basic renaming obfuscation has become widely-used among Chinese third-party market developers, while still not pervasive in Google Play market. Besides, malware authors put great efforts on more advanced code protection skills, like string encryption and reflections. Also, we provide the corresponding illustrations to enlighten developers to select the most suitable code protection methodologies and help researchers improve code analysis systems in the right direction.

Acknowledgement. We thank anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (NSFC) under Grant No. 61572415 and 61572481, Hong Kong S.A.R. Research Grants Council (RGC) Early Career Scheme/General Research Fund No. 24207815 and 14217816.

References

1. smartphone assistant. http://zhushou.360.cn/
2. Allatori. http://www.allatori.com/
3. Androguard. https://github.com/androguard/androguard
4. Anzhi. http://www.anzhi.com/
5. Appchina. http://www.appchina.com/
6. Backward compatibility for android applications. https://android-developers.googleblog.com/2009/04/backward-compatibility-for-android.html
7. DashO. https://www.preemptive.com/products/dasho/overview
8. Dexguard. https://www.guardsquare.com/en/dexguard
9. DexProtector. https://dexprotector.com/
10. Huawei appstore. http://appstore.huawei.com/
11. Java Cryptography Extension. http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html
12. n-gram. https://en.wikipedia.org/wiki/N-gram
13. Number of available applications in the Google Play Store from December 2009 to December 2017. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
14. ProGuard. http://proguard.sourceforge.net/
15. Shield4J. http://shield4j.com/
16. Virusshare. https://virusshare.com/
17. Virustotal. https://www.virustotal.com/
18. Wandoujia. https://www.wandoujia.com/
19. Xiaomi application store. http://app.mi.com/
20. Apvrille, A., Nigam, R.: Obfuscation in android malware, and how to fight back. Virus Bull. 1–10 (2014)
21. Balachandran, V., Tan, D.J., Thing, V.L.: Control flow obfuscation for android applications. Comput. Secur. **61**, 72–93 (2016)
22. Bichsel, B., Raychev, V., Tsankov, P., Vechev, M.T.: Statistical deobfuscation of android applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2016)
23. Calvet, J., Fernandez, J.M., Marion, J.: Aligot: cryptographic function identification in obfuscated binary programs. In: Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS) (2012)
24. Chen, K., Liu, P., Zhang, Y.: Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: Proceeding of the 36th International Conference on Software Engineering (ICSE) (2014)
25. Duan, Y., et al.: Things you may not know about android (un)packers: a systematic study based on whole-system emulation. In: Proceedings of 25th Annual Network and Distributed System Security Symposium (NDSS) (2018)
26. Faruki, P., Fereidooni, H., Laxmi, V., Conti, M., Gaur, M.S.: Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions. CoRR abs/1611.10231 (2016)

27. Freiling, F.C., Protsenko, M., Zhuang, Y.: An empirical evaluation of software obfuscation techniques applied to Android APKs. In: Tian, J., Jing, J., Srivatsa, M. (eds.) SecureComm 2014. LNICST, vol. 153, pp. 315–328. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23802-9_24
28. Gröbert, F., Willems, C., Holz, T.: Automated identification of cryptographic primitives in binary programs. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 41–60. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23644-0_3
29. Hoffmann, J., Rytilahti, T., Maiorca, D., Winandy, M., Giacinto, G., Holz, T.: Evaluating analysis tools for android apps: status quo and robustness against obfuscation. In: Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy (CODASPY) (2016)
30. Huang, H., et al.: Android malware development on public malware scanning platforms: a large-scale date-driven study. In: Proceeding of the 2016 IEEE International Conference on Big Data (BigData) (2016)
31. Li, L., Bissyandé, T.F., Octeau, D., Klein, J.: DroidRA: taming reflection to support whole-program analysis of android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA) (2016)
32. Li, M., et al.: LibD: scalable and precise third-party library detection in Android markets. In: Proceedings of the 39th International Conference on Software Engineering (ICSE) (2017)
33. Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth attacks: an extended insight into the obfuscation effects on Android malware. Comput. Secur. **51**, 16–31 (2015)
34. Matenaar, F., Wichmann, A., Leder, F., Gerhards-Padilla, E.: CIS: the crypto intelligence system for automatic detection and localization of cryptographic functions in current malware. In: Proceeding of the 7th International Conference on Malicious and Unwanted Software (MALWARE), 16–18 October 2012, Fajardo, PR, USA (2012)
35. Park, J., Kim, H., Jeong, Y., Cho, S., Han, S., Park, M.: Effects of code obfuscation on Android app similarity analysis. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. **6**(4), 86–98 (2015)
36. Preda, M.D., Maggi, F.: Testing Android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. J. Comput. Virol. Hacking Tech. **13**(3), 209–232 (2017)
37. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: evaluating Android anti-malware against transformation attacks. In: Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2013)
38. Shu, J., Li, J., Zhang, Y., Gu, D.: Android app protection via interpretation obfuscation. In: Proceeding of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC) (2014)
39. Suarez-Tangil, G., Dash, S.K., Ahmadi, M., Kinder, J., Giacinto, G., Cavallaro, L.: DroidSieve: fast and accurate classification of obfuscated Android malware. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY) (2017)
40. Wang, H., Guo, Y., Ma, Z., Chen, X.: WuKong: a scalable and accurate two-phase approach to Android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA), Baltimore, MD, USA, 12–17 July 2015 (2015)

41. Wang, Y., Rountev, A.: Who changed you? Obfuscator identification for Android. In: Proceedings of the 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft) (2017)
42. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of 7th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec) (2014)