SoftwareX 11 (2020) 100403

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

# Obfuscapk: An *open-source* black-box obfuscation tool for Android apps

Simone Aonzo *, Gabriel Claudiu Georgiu, Luca Verderame, Alessio Merlo

*DIBRIS, University of Genoa, Italy*

ARTICLE INFO

*Article history:*
Received 3 September 2019
Received in revised form 28 November 2019
Accepted 15 January 2020

*Keywords:*
Android
Obfuscation
Program analysis

ABSTRACT

Obfuscapk is an open-source automatic obfuscation tool for Android apps that works in a black-box fashion (i.e., it does not need the app source code). Obfuscapk supports advanced obfuscation features and has a modular architecture that could be straightforwardly extended to support new obfuscation techniques. This paper introduces the architecture, the main obfuscation techniques implemented in Obfuscapk, as well as the basics of the Obfuscapk CLI. Finally, the paper discusses an actual use-case for Obfuscapk, and an empirical assessment on the reliability of the tool on a set of 1000 "most downloaded" APKs from the Google Play Store.



## Code metadata

| | |
|---|---|
| Current code version | 1.1.2 |
| Permanent link to code/repository used for this code version | https://github.com/ElsevierSoftwareX/SOFTX_2019_275 |
| Legal Code License | MIT |
| Code versioning system used | git |
| Software code languages, tools, and services used | Written in Python, tested and working on Ubuntu 18.04, Docker image available. Tools needed: apktool, jarsigner and zipalign. |
| Compilation requirements, operating environments & dependencies | Python packages: pycryptodome, tqdm, virustotal-api, Yapsy. |
| Support email for questions | gabriel.georgiu@talos-sec.com |

## 1. Motivation and significance

Obfuscation is a *security through obscurity* technique that modifies the code in order to counteract automatic or manual code analysis. However, it is considered as a double-edged sword by the security community because both software developers and malware authors frequently use obfuscation. In fact, on the one hand, obfuscation keeps developers' competitors away from copying the code and makes it difficult for attackers to alter the regular flow of the software (e.g., *cracking*). On the other hand, it also helps malware authors to circumvent automated code analysis and manual inspection. As an example, consider a malware sample that is being recognized by anti-virus engines. In this case, the malware author needs to quickly build a variant of the original malware that could go undetected. Since creating a variant from scratch is time-consuming, obfuscating the code of the original malware is often considered a good compromise.

The challenges around obfuscation have attracted many researchers, especially in the field of mobile apps, given the astonishing growth of such markets. In the mobile world, especially the one focusing on the Android platform, obfuscation is rather common. Disassembling (or decompiling) and rebuilding an Android app is more straightforward w.r.t. other binary code, like, e.g., x86 executables. So far, most of the studies on Android app obfuscation focus on how: (i) to build reliable obfuscation techniques [1,2], (ii) obfuscation can be handled by state-of-art code analysis tools [3], (iii) to automatically deobfuscate the code [4], and (iv) developers actually adopt obfuscation nowadays [5,6]. The recent research works suggest that many developers are unable to apply advanced obfuscation techniques and that free off-the-shelf obfuscators support only basic obfuscation or are very difficult to configure. Furthermore, there are several academic works [7–14] that take advantage of machine learning techniques for malware detection; however, only few of them [15,16] take into

* Corresponding author.
*E-mail address:* simone.aonzo@gmail.com (S. Aonzo).

https://doi.org/10.1016/j.softx.2020.100403

consideration obfuscation in their models. It is also worth to point out that there is an increasing trend [17,18] on applying adversarial machine learning on mobile, with the aim to study how an attacker could modify an app in order to evade an existing ML model. However, there exist only a couple of tools (i.e., ADAM [19] and AAMO [20]) that allow to apply (some) obfuscation techniques *automatically*. Unfortunately, both of them were never updated since the year they were released, ADAM does not implement advanced obfuscation techniques, and the current version of AAMO does not seem to work correctly. To overcome the limitation of previous tools, we have developed **Obfuscapk**,[1] a free Python tool that is able to obfuscate compiled Android apps (i.e., without the need of the source code). Obfuscapk supports advanced obfuscation features (e.g., string encryption and native libraries encryption), and its modular architecture easily allows to add new obfuscation techniques by the community. We tested Obfuscapk on 1000 APKs among the most installed apps from the Google Play Store; our experiments indicate that Obfuscapk automatically generates obfuscated full working apps in the 83% of the cases. Obfuscapk aims at becoming a useful tool for both the developers' and the research communities. On the one hand, developers can use Obfuscapk in cooperation with ProGuard, which is the default optimizer and obfuscator included in the Android SDK and supported by the official Android Studio IDE. First, the developer uses the Android SDK with ProGuard to release the APK, and then she can apply more advanced obfuscation techniques through Obfuscapk. On the other hand, the research community on mobile security can apply Obfuscapk as a black-box obfuscation tool to apps and malware samples for several aims, like building or attacking a machine learning model, improving program analysis techniques w.r.t. obfuscation transformations, just to cite a few. Finally, each user can extend the current tool by adding her own or other obfuscation techniques at state of the art. Obviously, several commercial obfuscators exist. ProGuard [21] and Allatori [22] work at the source-code level, while the others like DashO [23], DexProtector [24], and Shield4J [25] directly work on APK files. Unfortunately, it is hard to compare them, since they typically do not offer an evaluation version, and they are closed-source. Still, Obfuscapk implements all the advanced obfuscation techniques declared by such proprietary obfuscators.

## 2. Supported obfuscation techniques

In this section, we describe the techniques supported by Obfuscapk, with a specific focus on their impact on malware detection. There exists a classification [20,26] of obfuscation techniques for the Android ecosystem, which divides the techniques in two main categories: trivial and non-trivial.

### 2.1. Trivial techniques

Trivial techniques are the simplest ones. They have no real obfuscation effects on the APK, but they can trick some signature-based anti-malware tools [26]. Obfuscapk implements four existing trivial techniques, namely: **Align**, **Re-sign**, **Rebuild**, and **Randomize Manifest**.

[1] Android Package (APK) is the package file format used by the Android operating system for distribution and installation of mobile apps.

#### 2.1.1. Align and re-sign

These techniques implement the last mandatory steps for building a working Android APK. The alignment is done by using `zipalign`, a specific tool of the Android SDK. The result is a reorganized application in which the structure of the files is optimized for running on an Android device. Android requires all APKs to be digitally signed with a certificate before they can be installed on a device (or updated), so the Re-sing step is the last mandatory step after applying obfuscation.

#### 2.1.2. Rebuild

The bytecode contained in `classes.dex`[2] file can be disassembled and reassembled to obtain a different version of the file. Such technique transforms the bytecode without changing its semantic, in order to preserve the original behavior of the app. This rebuild aims to fool anti-malware tools that use the signature of *classes.dex* file.

#### 2.1.3. Randomize manifest

Such technique randomly rearranges the entries in the `AndroidManifest.xml`,[3] without modifying the XML tree structure. The goals are twofold: (i) change the hash of the manifest file, and (ii) fool the N-gram analysis [27].

### 2.2. Non-trivial techniques

Non-trivial techniques are more complex, but they grant a more profitable gain in terms of detection rate and robustness [26]. The targets of obfuscation are both bytecode and resources (XMLs, asset files, and external libraries). Non-trivial obfuscation techniques can be divided into four subcategories: **Renaming**, **Encryption**, **Code**, and **Resources**.

#### 2.2.1. Renaming

In software development, the names of identifiers (variable names, function names, and so forth) should be meaningful to provide good code readability and maintainability. However, such clear names may leak information about code functionalities. Furthermore, since the package name uniquely identifies an Android app, its modification amounts to put a new app in the Android ecosystem. Therefore, the renaming technique substitutes each identifier with an obscure and meaningless one. While the methods and fields renaming has no drawbacks, the classes and package name renaming is more complicated because the `AndroidManifest.xml` must be updated accordingly.

#### 2.2.2. Encryption

An APK file may contain resources that can be requested at run-time by the developer. Those files can be native libraries or even strings. Such resources can be encrypted and decrypted at run-time. In this case, the attacker needs another step to find the decryption key before reading the resources, but there is also an obvious disadvantage: the app performances get worse because it needs extra calculations when it accesses its resources. When Obfuscapk starts, it automatically generates a random secret key (32 characters long, using ASCII letters and digits) that can be used to encrypt:

- **LibEncryption**. Native libraries;
- **AssetEncryption**. Asset files (like videos, photos, text files, etc.);
- **ResStringEncryption**. Strings contained in the `strings.xml` resource file.
- **ConstStringEncryption**. Constant strings in the code.

[2] The `classes.dex` file contains the Android Dalvik bytecode, i.e. the application code.

[3] The `AndroidManifest.xml` file contains the information of the app, like components and permissions.

#### 2.2.3. Code

This category contains all obfuscation techniques that affect instructions inside the `classes.dex`. There exist several techniques that hide the behavior of the application, each of which is applied to a different aspect of the code.

**DebugRemoval**. This technique just removes debug meta-data. The removal of debug information, such as line numbers, types, or method names, reduces the amount of useful information for the reverse engineering process.

**CallIndirection**. This technique modifies the control-flow graph (CFG from now on) without impacting the code semantics; it adds new methods that invoke the original ones. For example, an invocation to the method $m_1$ will be substituted by a new wrapper method $m_2$, that, when invoked, it calls the original method $m_1$.

**Goto**. Given a method, it inserts a `goto` instruction pointing to the end of the method and another `goto` pointing to the instruction after the first `goto`; it modifies the CFG by adding two new nodes.

**Reorder**. This technique consists of changing the order of basic blocks in the code. When a branch instruction is found, the condition is inverted (e.g., "branch if lower than", becomes "branch if greater or equal than") and the target basic blocks are reordered accordingly. Furthermore, it also randomly re-arrange the code abusing `goto` instructions.

**ArithmeticBranch**. This is the first technique that belongs to the *junk code insertion* category, that aims at adding some useless and semantic-preserving instructions to the code. In this case, the junk code is composed by arithmetic computations and a branch instruction depending on the result of these computations, crafted in such a way that the branch is never taken.

**Nop**. Nop, short for *no-operation*, is a dedicated instruction that does nothing. This technique just inserts random `nop` instructions (i.e., junk code) within every method implementation.

**MethodsOverload**. It exploits the overloading feature of the Java programming language to assign the same name to different methods but using different arguments. Given an already existing method, this technique creates a new void method with the same name and arguments, but it also adds new random arguments. Then, the body of the new method is filled with random arithmetic instructions.

#### 2.2.4. Invocation by reflection

The Reflection is a feature of the Java programming language that allows examining or modifying the run-time behavior of a class during execution. In this context, this feature is used to invoke methods of a given object.

**Reflection**. This technique analyzes the existing code looking for method invocations of the app, ignoring the calls to the Android framework. If it finds an instruction with a suitable method invocation (i.e., no constructor methods, public visibility, enough free registers, …) such invocation is redirected to a custom method that will invoke the original method using the Reflection APIs.

**AdvancedReflection**. This technique is complementary to the previous one because it works in the same way, but it targets the invocations of dangerous APIs. In order to find out if a method belongs to the Android Framework, Obfuscapk refers to the mapping discovered by Backes et al. [28].

### 2.3. Summary of techniques by categories

Table 1 summarizes the aforementioned techniques implemented in Obfuscapk, grouped by categories.

## 3. Software description

### 3.1. Software architecture

Obfuscapk is designed (see Fig. 1) to be modular and easy to extend, so it is built on *Yapsy*, a plugin management system. Consequently, each obfuscator is a plugin that inherits from an abstract base class and needs to implement the method `obfuscate`. When the tool begins to process an APK, it creates an `obfuscation` object to store all the needed information (i.e., the location of the decompiled code) and the internal state of the operations (i.e., the list of already used obfuscators). Then, the `obfuscation` object is passed, as a parameter to the obfuscate method, to all the active plugins/obfuscators sequentially. The list and the order of the active plugins are specified through command-line options.

The tool is easily extensible with new obfuscators: it is enough to add the source code implementing the obfuscation technique and the plugin metadata (a `<obfuscator-name>.obfuscator` file) in the `src/obfuscapk/obfuscators` directory. The tool will automatically detect the new plugin, with no need of further configuration steps.

### 3.2. Tool functionalities

The complete set of Obfuscapk functionalities is provided by the following help message:

```
obfuscapk [-h] -o OBFUSCATOR [-w DIR] [-d OUT_APK]
          [-i] [-p] [-k VT_API_KEY]
          <APK_FILE>
```

There are two mandatory parameters: `<APK_FILE>`, the path (relative or absolute) to the apk file to obfuscate and the list with the names of the obfuscation techniques to apply (specified with the `-o` option). The remaining parameters are optional.

- `-o` the list with the names of the obfuscation techniques (previously described in Section 2 and summarized in Table 1) to apply; e.g., `-o Rebuild -o NewSignature -o NewAlignment`.
- `-w DIR` is used to set the working directory used to store the intermediate files (generated by apktool). If not specified, a directory named `obfuscation_working_dir` is created in the same directory of the input app. This option can be useful for debugging purposes.
- `-d OUT_APK` is used to set the path of the destination file, i.e., the apk file generated by the obfuscation process. If not specified, the final obfuscated file will be saved inside the working directory. Existing files will be overwritten without any warning.
- `-i` is a flag for ignoring known third party libraries during the obfuscation. This option could be useful to improve performances and reduce the risk of errors. The list of libraries to ignore is obtained from the LiteRadar[4] project.
- `-p` is a flag for showing progress bars during the obfuscation operations.
- `-k VT_API_KEY` is only needed when using VirusTotal obfuscator, to set the API key(s) to be used when communicating with Virus Total. It can be set multiple times to cycle through the API keys during the requests (e.g., `-k VT_KEY_1 -k VT_KEY_2`).

[4] https://github.com/pkumza/LiteRadar

**Table 1**
Obfuscapk implemented obfuscators.

| Category | Obfuscapk obfuscator |
|---|---|
| Trivial | `RandomManifest`, `Rebuild`, `NewAlignment`, `NewSignature` |
| Renaming | `ClassRename`, `FieldRename`, `MethodRename` |
| Encryption | `LibEncryption`, `ResStringEncryption`, `AssetEncryption`, `ConstStringEncryption` |
| Code | `ArithmeticBranch`, `Reorder`, `CallIndirection`, `DebugRemoval`, `Goto`, `MethodOverload`, `Nop` |
| Reflection | `Reflection`, `AdvancedReflection` |

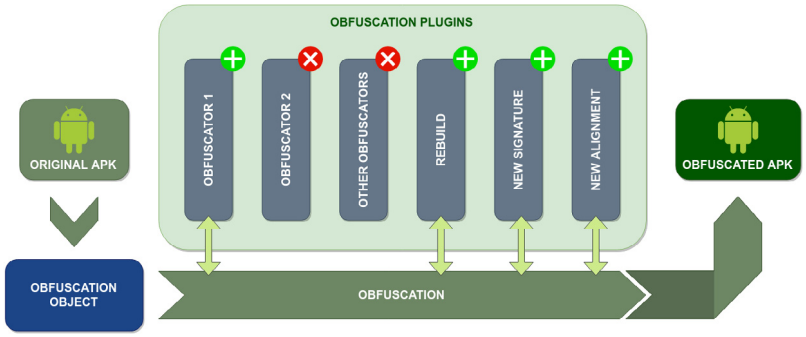


**Fig. 1.** Obfuscapk architecture.

**Table 2**
Detection ratio of different obfuscated versions.

| Category | Detection ratio | Percentage |
|---|---|---|
| Original | 32/58 [29] | 55% |
| Trivial | 18/58 [31] | 31% |
| Renaming | 16/58 [32] | 28% |
| Reflection | 15/58 [33] | 26% |
| Code | 8/58 [34] | 14% |
| Encryption | 0/58 [35] | 0% |

## 4. Illustrative example

As an example, we use Obfuscapk to obfuscate an Android malware discovered in early 2019, a Trojan-Banker named *CometBot*. In this example, we have obfuscated our specimen [29] using the different sets of techniques implemented in Obfuscapk (summarized in Table 1). Then, we have uploaded the obfuscated sample to Virus Total [30]; results are reported in Table 2, ordered by detection ratio.

This example shows how different types of obfuscation influence, more or less, the detection ratio. In this particular case, the techniques of the Encryption category allowed to build an *undetectable variant*. Since an Android APK is an archive that contains several files and a malicious component might be implemented almost everywhere, it is not possible to establish which is the most effective subset of techniques *a priori*, since each technique has different effects on the files within the APK.

Listing 1 shows the command line parameters for obfuscating the CometBot malware using the Encryption techniques.

**Listing 1:** Obfuscating Cometbot malware using encryption

```
$ obfuscapk \
-o LibEncryption -o ResStringEncryption \
-o AssetEncryption -o ConstStringEncryption \
-o Rebuild -o NewAlignment -o NewSignature \
-d encryption.apk cometbot.apk
```

## 5. Testing the stability of obfuscapk

We empirically evaluated the stability of Obfuscapk by obfuscating, using each implemented technique, a dataset of 1000 APKs randomly downloaded from the Google Play Store, among the top free apps by the number of installations [36]. Then, we have tested if the modified APK can be still installed and if it runs properly.

A single automated test, for each APK $A$, executes as follow:

1. Obfuscapk obfuscates $A$, and builds up a new APK $A'$.
2. $A'$ is installed on an actual Android device.
3. If the above step fails, the original $A$ is installed, in order to verify whether the failure is due to the modification carried out by Obfuscapk or it is independent.
4. If the installation of $A'$ has been successful, its behavior is tested by generating a stream of 1024 pseudo-random user events with *Monkey* [37], seeded by a random number $n$. Using different seed values leads to generate distinct sequences of user events. If $A'$ fails, this can be due either to Obfuscapk transformations or to the presence of bugs in the original app $A$.
5. To discriminate, we stimulate $A$ with the same stream of events, generated by seeding *Monkey* with the same seed $n$.

Through previous steps, we can empirically assess whether the obfuscation process leads to failures. We carried out the experimental assessment on a Dell XPS 9530 (Ubuntu 18.04, Intel Core i7-4712HQ @ 2.30 GHz, 16GB RAM), as well as on a OnePlus 6 (Android 9, Snapdragon 845 @ 2.8 GHz, 8GB RAM) that we used to obfuscate, install and (automatically) stimulate the apps. We emphasize that the results of the same sequences of Monkey events on the original and obfuscated version of the app under test are equivalent, because: (i) we use the same seed, thereby obtaining the same sequence of events (ii) Obfuscapk does not affect the layout of the app defined in the related XML files (buttons, editable fields, etc.) (iii) when Obfuscapk renames some component, if necessary, it modifies every reference in the manifest file accordingly. In this way, the event generated

by Monkey triggers the same component in the original and in the obfuscated app, while the code of the obfuscated app is semantically equivalent to the code in the original one.

### 5.1. Stability results

Our results indicate that on 1000 samples, 47 repackaged apps failed at Step 2., i.e., they could not be installed successfully. Among these, 8 also failed Step 3., meaning that the corresponding original APKs were already broken in some way. Therefore, we discarded them, and we considered a new set, consisting of the original samples without the broken APKs; that is, $1000 - 8 = 992$ apps. On this set, the 95% (940/992) of repackaged apps have been successfully installed. Then, we stimulated the installed apps according to 4.. Among these, 817 were repackaged and stimulated without crashes, while $123 = 940 - 817$ failed and required further analysis. Therefore, we applied Step 5. to such apps obtaining that 6 original apps crashed, thereby proving that the same problems affected the original app, too. For this reason, we also discarded these APKs, reaching a total of $992 - 6 = 986$ of working original apps. Summing up, the $83\% = 817/986$ of working apps have been successfully obfuscated, installed, and executed properly after the obfuscation process. It is worth pointing out that *Monkey* [37] is not a comprehensive tool for dynamically testing Android APK; therefore, such 83% must be considered an upper-bound. We discourage the use of Monkey to test the stability of an app in production.

Then, we tried to classify the reasons behind the $17\% = 169/986$ failures. First of all, Obfuscapk relies on Apktool to decompile and rebuild the target APK. In order to understand if an APK crashed because of a malfunction of Apktool, we have repeated our experiments just using the `Rebuild`, `NewAlignment` and `NewSignature` components of Obfuscapk. In this way, Obfuscapk does not make any change to target APK, but it just rebuilds the app using Apktool (this requires to align and sign the APK to install it). The $66\% = 111/169$ failed. Unfortunately, using a black-box approach (i.e., without having the source code of the app), it is unfeasible to discriminate between a failure of the Apktool rebuild procedure or a failure originated by an anti-tampering technique implemented by the original app. Instead, the remaining $34\% = 58/169$ fail due to the modifications carried out by Obfuscapk. Unfortunately, the reasons for the failure are heterogeneous and APK-specific. Still, we noticed that the most common failures are related to the use of reflection, i.e., when Obfuscapk renames a class or method, and the app tries to load it using the original name.

## 6. Impact and conclusions

Obfuscapk is a black-box obfuscation tool that can work with every Android APK and offers a free solution with advanced obfuscation techniques. Users can extend Obfuscapk with newly studied obfuscation techniques, or they can improve the existing ones. Furthermore, Obfuscapk can be adopted to study the effects of obfuscation transformations with state of the art on program analysis.

It is worth noting that obfuscated versions of the same app may be effectively uploaded on the Play Store, substituting the existing release. The only constraint is that the package name remains unchanged because it is used as an unique identifier. To the best of our knowledge Obfuscapk does not violate any rules of the Play Store.

In the foreseeable future, we hope that a community of developers and users around Obfuscapk can grow, so that the tool can improve in terms of both features and stability, in order to become a research and industrial reference implementation for the obfuscation of Android apps.

## Declaration of competing interest

We wish to confirm that there are no known conflicts of interest associated with this publication, and there has been no financial support for this work that could have influenced its outcome.

## Acknowledgment

This work has been partially supported by a grant of the Italian Presidency of Ministry Council.

## References

[1] Apvrille A, Nigam R. Obfuscation in android malware, and how to fight back. Virus Bull 2014;1–10.
[2] Shu J, Li J, Zhang Y, Gu D. Android app protection via interpretation obfuscation. In: 2014 IEEE 12th international conference on dependable, autonomic and secure computing. IEEE; 2014, p. 63–8.
[3] Rastogi V, Chen Y, Jiang X. Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM; 2013, p. 329–34.
[4] Bichsel B, Raychev V, Tsankov P, Vechev M. Statistical deobfuscation of android applications. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM; 2016, p. 343–55.
[5] Dong S, Li M, Diao W, Liu X, Liu J, Li Z, Xu F, Chen K, Wang X, Zhang K. Understanding android obfuscation techniques: a large-scale investigation in the wild. In: International conference on security and privacy in communication systems. Springer; 2018, p. 172–92.
[6] Wermke D, Huaman N, Acar Y, Reaves B, Traynor P, Fahl S. A large scale investigation of obfuscation use in google play. In: Proceedings of the 34th annual computer security applications conference. ACM; 2018, p. 222–35.
[7] Wu D-J, Mao C-H, Wei T-E, Lee H-M, Wu K-P. Droidmat: android malware detection through manifest and api calls tracing. In: Information security (Asia JCIS), 2012 seventh asia joint conference on. IEEE; 2012, p. 62–9.
[8] Aung Z, Zaw W. Permission-based android malware detection. Int J Sci Technol Res 2013;2(3):228–34.
[9] Sanz B, Santos I, Laorden C, Ugarte-Pedrero X, Bringas PG, Álvarez G. PUMA: permission usage to detect Malware in Android. In: International joint conference CISIS'12-ICEUTE´ 12-SOCO´ 12 special sessions; 2013.
[10] Liu X, Liu J. A two-layered permission-based android malware detection scheme. In: Mobile cloud computing, services, and engineering (mobilecloud), 2014 2nd ieee international conference on. IEEE; 2014, p. 142–8.
[11] Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K. DREBIN: effective and explainable detection of android malware in your pocket. In: NDSS. 2014.
[12] Ban T, Takahashi T, Guo S, Inoue D, Nakao K. Integration of multi-modal features for android malware detection using linear SVM. In: Information security (AsiaJCIS), 2016 11th asia joint conference on. IEEE; 2016, p. 141–6.
[13] Idrees F, Rajarajan M, Conti M, Chen TM, Rahulamathavan Y. Pindroid: a novel android malware detection system using ensemble learning methods. Comput Secur 2017;68:36–46.
[14] Aonzo S, Merlo A, Migliardi M, Oneto L, Palmieri F. Low-resource footprint, data-driven malware detection on android. IEEE Trans Sustain Comput 2017.
[15] Suarez-Tangil G, Dash SK, 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. ACM; 2017, p. 309–20.
[16] Garcia J, Hammad M, Pedrood B, Bagheri-Khaligh A, Malek S. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Tech. Rep, Department of Computer Science, George Mason University; 2015.
[17] Demontis A, Melis M, Biggio B, Maiorca D, Arp D, Rieck K, Corona I, Giacinto G, Roli F. Yes, machine learning can be more secure! a case study on android malware detection. IEEE Trans Dependable Secure Comput 2017.
[18] Chen X, Li C, Wang D, Wen S, Zhang J, Nepal S, Xiang Y, Ren K. Android hiv: a study of repackaging malware for evading machine-learning detection. 2018, arXiv preprint arXiv:1808.04218.
[19] Zheng M, Lee PP, Lui JC. ADAM: an automatic and extensible platform to stress test android anti-virus systems. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer; 2012, p. 82–101.

[20] Dalla Preda M, Maggi F. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. J Comput Virology Hacking Tech 2017;13(3):209–32.
[21] ProGuard. https://www.guardsquare.com/en/proguard. [Accessed January 24, 2020].
[22] Allatori. http://www.allatori.com. [Accessed January 24, 2020].
[23] DashO. https://www.preemptive.com/products/dasho/overview. [Accessed January 24, 2020].
[24] DexProtector. https://dexprotector.com. [Accessed January 24, 2020].
[25] Shield4J. http://shield4j.com. [Accessed January 24, 2020].
[26] Maiorca D, Ariu D, Corona I, Aresu M, Giacinto G. Stealth attacks: an extended insight into the obfuscation effects on android malware. Comput Secur 2015;51:16–31.
[27] Shabtai A, Fledel Y, Elovici Y. Automated static code analysis for classifying android applications using machine learning. In: 2010 international conference on computational intelligence and security. IEEE; 2010, p. 329–33.
[28] Backes M, Bugiel S, Derr E, McDaniel P, Octeau D, Weisgerber S. On demystifying the android application framework: re-visiting android permission specification analysis. In: 25th {USENIX} security symposium ({USENIX} Security 16). 2016, p. 1101–18.
[29] CometBot original. https://www.virustotal.com/gui/file/642da73bc4c78004304dfed2e6e704ebb352ff9f1db19a19cc2296c86164e723. [Accessed January 24, 2020].
[30] Virus Total. https://www.virustotal.com. [Accessed January 24, 2020].
[31] CometBot – obfuscated with trivial techniques. https://www.virustotal.com/gui/file/1fe6ad3bd534bf9f42cbdefa66e99db1760bb110d978dfb28517bd61fb5e9a16. [Accessed January 24, 2020].
[32] CometBot – obfuscated with renaming.https://www.virustotal.com/gui/file/e58332461b8151e842369a635fa01822289f45128c5d5afcc981c7cb2ba170d4. [Accessed January 24, 2020].
[33] CometBot – obfuscated with reflection. https://www.virustotal.com/gui/file/feddd8ac2ce246105c5df050061ea5dad8cb5da8411010646f3f9eb8dbbc1b44. [Accessed January 24, 2020].
[34] CometBot – obfuscated with code manipulation. https://www.virustotal.com/gui/file/356a5c92670b825d0bf3e2e927ce3f2ff3a407ad1b6e91119a8056391e665b0c. [Accessed January 24, 2020].
[35] CometBot – obfuscated with encryption. https://www.virustotal.com/gui/file/cc394ba746f55630d97f06df89d851438c866c6179e39eb5d706969ca7a40de0. [Accessed January 24, 2020].
[36] ANDROIDRANK – open android market data. https://www.androidrank.org/app/ranking?price=free. [Accessed January 24, 2020].
[37] UI/application exerciser monkey. https://developer.android.com/studio/test/monkey.html. [Accessed January 24, 2020].