An Empirical Study of Code Deobfuscations on Detecting Obfuscated Android Piggybacked Apps

Yanxin Zhang*, Guanping Xiao⋆, Zheng Zheng†, Tianqing Zhu*, Ivor W. Tsang*, Yulei Sui*
*School of Computer Science, University of Technology Sydney, Australia
⋆College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
†School of Automation Science and Electrical Engineering, Beihang University, China
yanxin.zhang@student.uts.edu.au, gpxiao@nuaa.edu.cn, zhengz@buaa.edu.cn, {tianqing.zhu, ivor.tsang, yulei.sui}@uts.edu.au

***Abstract*—Android piggybacked malware (i.e., apps that piggyback malicious code) are becoming ubiquitous in app stores. Malware writers often use obfuscation techniques to obfuscate piggybacked apps to evade detection by Android malware detectors. Previous studies in this field have focused on the impact of code obfuscations on the detection of piggybacked malware, but the impact of code deobfuscation on detecting obfuscated piggybacked apps has rarely been studied. Knowing about the impact of code deobfuscation can provide useful insights into obfuscated piggybacked apps and therefore the design of resilient Android malware detectors. In this paper we conduct an empirical study of code deobfuscations on detecting obfuscated Android piggybacked apps, focusing on three types of malware detectors: commercial anti-malware products, machine learning-based detectors, and similarity-based detectors. We observe that code deobfuscations can impact differently depending on the malware detectors. For example, some deobfuscation strategies can improve the precision of detecting obfuscated piggybacked apps. Also we observe that the examined deobfuscation tools (Simplify and Deguard) have a different impact on obfuscated piggybacked apps after deobfuscations.**

I. Introduction

As the world's most popular mobile platform, the Android operating system (OS) impacts significantly on many people's daily life, thus the interest in its security and security enhancement is gaining increasing momentum in areas from academia to industry [1, 2]. Android apps are written in Java programming language, making them easy to be reversed by development tools. For example, the Apktool is often used to disassemble executable code and decode resource files of Android apps [3]. Since Android allows self-signed certificate apps, once an app is disassembled and decoded, its code and resource files can be modified, resigned, and repackaged as a new app. Worse, users feel free to install these unofficially released apps, resulting in potential security issues.

Malware developers usually unpack benign and preferably popular apps for piggybacking code fragments with malicious behaviours, as illustrated in Fig. 1. These apps are referred to as piggybacked apps, and are widely available in real-world markets, which is seriously threatening user security and destroying the reputation of the developers of the original apps. Zhou *et at.* [4] pointed out that 86% of the 1,260 malicious application samples examined are piggybacked malicious apps. To protect users against these malicious apps, various anti-malware detectors, such as VirusTotal and SimiDroid, have been released to detect malware [5].

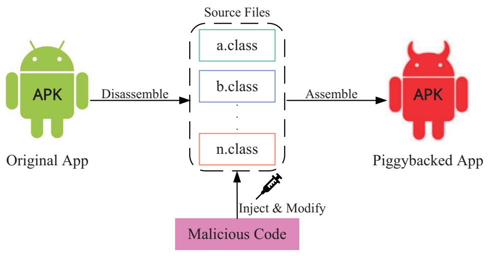


Fig. 1. Illustration of Android piggybacked malware.

Unfortunately, piggybacked app developers normally use code obfuscations to evade such detection. Code obfuscation is a reorganization and reprocessing technology used on previously released programs [6]. An obfuscation technique transforms the original program into a new one while maintaining its functionality, but the obfuscation code makes the decompilation difficult to figure out the true semantics of the original program. Recent studies have shown that code obfuscations used in piggybacked apps have caused a significant decrease in the ability to accurately detect Android malware [7, 8].

To overcome the challenge of detecting obfuscated Android malware, researchers have developed deobfuscation tools to identify obfuscated codes injected into apps by malware writers. For example, Simplify [9], developed by Caleb Fenton, uses a virtual machine sandbox for executing an app to understand its behavior. Then, it tries to optimize the code so that the decompiled code is simplified and easier for humans to understand. Different from the work by Fenton, Bichsel *et al.* [10] developed Deguard, a statistical deobfuscation tool for Android. Deguard phrases the layout deobfuscation problem of Android apps as a structured prediction in a probabilistic graphical model for identifiers that are based on the occurrence of names.

However, there is little empirical evidence on the impact of code deobfuscations on detecting obfuscated Android piggybacked apps. Such empirical knowledge can provide useful insights for researchers and security engineers to better understand the influences of obfuscation strategies and deobfuscation tools on Android malware detectors. Thus, such evidence can guide the design of resilient and effective Android malware detectors. Therefore, in this paper we perform a large-scale empirical study of the impact of code deobfuscations on detecting obfuscated Android piggybacked apps.

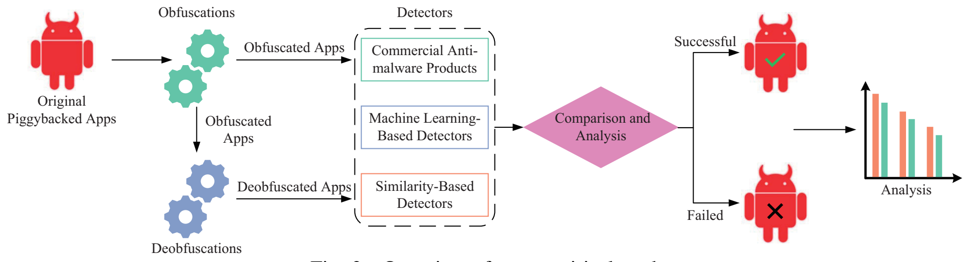


Fig. 2. Overview of our empirical study.

Fig. 2 shows an overview of our empirical study. First, we generated obfuscated piggybacked apps using 1,399 pairs of original piggybacked apps from [11] by utilizing different obfuscation strategies. The generated obfuscated apps were later deobfuscated to generate deobfuscated piggybacked apps. To present a thorough analysis, we used three types of Android malware detectors, including commercial anti-malware products, machine learning-based detectors, and similarity-based detectors, as the targeted systems for detecting the generated obfuscated/deobfuscated piggybacked apps. Finally, having used 10 different strategies, we analyzed the precision of these detectors to understand the impact of code deobfuscations.

This paper makes the following main contributions:

- To the best of our knowledge, this is the first work to study the impact of code deobfuscations on detecting obfuscated Android piggybacked apps.
- We have generated 11,378 obfuscated and deobfuscated piggybacked apps from 1,399 original piggybacked apps using 10 strategies.
- We have evaluated the precision of three types of Android malware detectors, including commercial anti-malware products (i.e., VirusTotal), machine learning-based detectors (i.e., Drebin and CSBD), and similarity-based detectors (i.e., Androguard and SimiDroid) on the generated obfuscated and deobfuscated piggybacked apps.

II. Background

A. Piggybacked Android Malware

Piggybacking refers to the behaviour that an unauthorized developer adds, deletes, or makes no changes to the non-core code of a benign Android app and signs the packaging with an unauthorized key [11]. Piggybacked apps have two characteristics. First, a piggybacked app is similar to the original application's core code, referring to the part after removing external common code (e.g., third-party libraries). This is because a piggybacked app needs to keep the function of the origin app. Also, the hacked app must be resigned, and this signature is generally not consistent with the original signature (e.g., the information of the developers).

B. Obfuscation and Deobfuscation of Android Apps

Code obfuscation doesn't make the code undecompiled. On the contrary, it conducts transformation of the code on structured code features (e.g., modifying program's control-flows) or on unstructured code features (e.g., renaming the apps' classes, methods, variables to meaningless names). In this paper, we use four types of obfuscation strategies, i.e., control flow flattening, insert junk code, identifier obfuscation, and string obfuscation, as shown in Table I.

Control Flow Flattening (CFF). This type of obfuscation strategy represents that the control statements in java code (e.g., "if", "while", "for", and "do"), which are converted into switch branch statements without changing the function of the source code. The advantage of CFF is that it blurs the relationship between the code blocks in the switch, increasing the difficulty of analysis. This technique divides the method into multiple basic blocks (case code blocks) and an entry block and allows these basic blocks to have both a common predecessor module and successor module to achieve flattening. The predecessor module mainly performs the distribution of basic blocks, and the distribution is realized by changing the switch variable. The successor module is used to update the value of the switch variable and jump to the beginning of the switch.

Insert Junk Code (IJC). This type of obfuscation strategy inserts a set of useless bytes into an original program without changing the original logic of the program. The program will still run normally, but the disassembly tool will fail to disassemble the inserted bytes, e.g., in a manner that the disassembly tool reports an error when the first few bytes of the normal instruction being recognized as an incomplete instruction. Therefore, the inserted instructions are random and incomplete. Note that these instructions must satisfy two conditions. Firstly, the instructions are located in a path that will never be executed when the program is running, and secondly that these instructions are part of the legal instructions except that they are incomplete instructions.

Identifier Obfuscation (IO). Identifier obfuscation is to rename the packages, classes, methods, and variables in a source program, and then replace them with meaningless identifiers, making it harder to crack the identifiers for analysis.

String Obfuscation (SO). To avoid the disassembled code being easy to analyze and understood, more critical string variables in the source program are often obfuscated. There are two kinds of string obfuscation strategies, namely encoding obfuscation and string encryption. Encoding obfuscation first converts the string into a hexadecimal array or Unicode encoding, and then restores it to a string when the string is called. After encoding obfuscation takes place, a series of numbers or garbled characters, which are difficult to analyze directly, exist in the reverse direction. String encryption is local encryption of the string, and then encode the ciphertext into the source code. Furthermore, implementing a decryption function as well as calling the decryption function decrypts the ciphertext where it can be used.

Code deobfuscation is a reverse process of code obfuscation. The code deobfuscation usually deobfuscates the obfuscated according to three key aspects, as follows: (1)

TABLE I
OBFUSCATION STRATEGIES

Type	Strategy	Description
CFF	API_REFLECTION	Hides all APIs by Java reflections.
	API_INS	Inserts invalid APIs between existing APIs.
IJC	BYTECODE	Inserts useless codes in the source code, such as defining a useless method, passing the class variable in, then not processing or shifting the useless codes, which looks like complicated codes, but actually same function.
IO	VARIABLE	Renames the variable name in the source code, then replaces it with a meaningless identifier, making it harder to crack this analysis.
	PCM	Obfuscate the package name in the same way as the variable name.
	BENIGN_CLASS	Obfuscate the class name in the same way as the variable name.
	RESOURCE_IMAGE	Modifies directly at the source level, then replaces the code and renames "icon.png" to "a.png", lastly hands it over to Android for compilation.
	RESOURCE_XML	Modifies directly at the source level, then replaces the code and "R.string.name" in xml file with "R.string.a", and hands it over to Android for compilation.
SO	STRING	Encrypts the string locally, then hardcodes the ciphertext into it. Lastly, decrypts it at runtime.
	BEN_PERMISSION	Encrypts the permission locally, hardcodes the ciphertext into it, and decrypts it at runtime.

readability. The purpose of deobfuscation is to make the code readable. The simpler the code, the more readable it is; (2) *better understanding of program's control- and data-flows*. This helps us to analyze the possible execution flow of the program statically; and (3) *getting context*. the contextual relevance of the program can help us better understand the semantic of a program. Several attempts working on the code deobfuscation of Android apps have been conducted. DeGuard [10] was proposed to deal with layout obfuscation introduced by ProGuard. The key idea is to summarize a probabilistic model by learning unobfuscated apps on a large scale and then use the model to recover the obfuscated code. Also, Baumann *et al.* [12] used a similar approach to perform ProGuard deobfuscation by code matching.

III. RESEARCH METHODOLOGY

A. Dataset and App Generation

1) Original Apps: We use the dataset, i.e., 1,497 pairs of original/piggybacked Android apps, which were collected from the work conducted by Li *et al.* [11]. In the work, the authors first sent all apps to VirusTotal to collect their associated anti-virus scanning reports. Then, based on the results of VirusTotal, they classified the set of apps into two subsets: one containing only benign apps and the other containing only malicious apps. Then they compared all the malicious apps and the benign apps according to the identity packages. If the pair of apps that were compared had different authors, they continued the comparison to establish whether the SDK version is the same. Finally, they collected the dataset of 1,497 app pairs, where each pair include an original benign app and a piggybacked app with a malicious payload. According to the provided hash values, we downloaded all 1,497 pairs of original/piggybacked Android apps.

2) Obfuscated Piggybacked Apps: Among the 1,497 app pairs, there are some duplicates after our manual inspection. The dataset has several benign apps corresponding to the same piggybacked app. After filtering these duplicate piggybacked apps, we obtained validated 1,399 pairs for our study. In order to understand the impact of each obfuscation strategy, we used AVPASS to obfuscate the validated 1,399 apps pairs with 10 obfuscation strategies (Table I). Table II shows the number of obfuscated piggybacked apps generated for each obfuscation strategy. Note that the numbers of obfuscated piggybacked apps can be different for each strategy, since not all the 1,399 piggybacked apps can be successfully obfuscated by the obfuscation strategies. For example, only 444 piggybacked apps were successfully obfuscated using BYTECODE.

3) Deobfuscated Piggybacked Apps: To create deobfuscated piggybacked apps, we leveraged two general Android deobfuscator, i.e., Simplify [9] and Deguard [10].

Simplify. The design of Simplify is inspired by dex-oracle, in which the Dalvik virtual machine is simulated, and the code is decompiled once executed [9]. After learning the function, the decompiled code is simplified into a form that the analyst can understand. Simplify includes two modules, i.e., Smalivm and Simplify. Smalivm is the simulator module of the Dalvik virtual machine and is mainly used for executing Dalvik virtual machine, based on the input smali file, returns all possible execution paths. The simplify module is the main module used to solve the obfuscation, which is primarily based on the analysis results of Smalivm. It simplifies the obfuscated decompiled code and generates the easy-to-understand decompiled code. We used Simplify to generate the deobfuscated piggybacked apps, using the obfuscated apps (Table II). The number of deobfuscated piggybacked apps by Simplify is shown in Table III.

Deguard. DeGuard is a new system for statistical deobfuscated Android APKs [10]. It deobfuscates an APK with large-scale learning and then summarizes a probability model to identify the code through its probability model. Using these models, DeGuard restores important information in the Android APK, including method and class names, as well as third-party libraries. Deguard then reveals string decoders and classes that handle sensitive data in Android malware. The process is divided into three steps: (1) Generating a dependency graph, where each node represents the element to be renamed, and each line represents a dependency; (2) Exporting restriction rules, which guarantees the APK being replied is a normal APK and keeping the same semantics as the original APK; (3) Predicting and recovering the original name of an obfuscated element according to the weighting provided by the probability model.

We wrote a python script with selenium [13] in order to use Deguard. Python script automatically opens the official

TABLE II
NUMBER OF OBFUSCATED APPS

Strategy	Number	Strategy	Number
STRING	1,130	BE_CLASS	1,372
VARIABLE	1,148	API_INS	933
PCM	1,311	BEN_PER	1,369
BYTECODE	444	API_REF	1,282
RESOURCE_IMAGE	1,370	RESOURCE_XML	1,370

TABLE III
NUMBER OF DEOBFUSCATED APPS (SIMPLIFY)

Strategy	Number	Strategy	Number
STRING	1,130	BE_CLASS	1,370
VARIABLE	1,148	API_INS	933
PCM	1,311	BEN_PER	1,369
BYTECODE	444	API_REF	933
RESOURCE_IMAGE	1,370	RESOURCE_XML	1,370

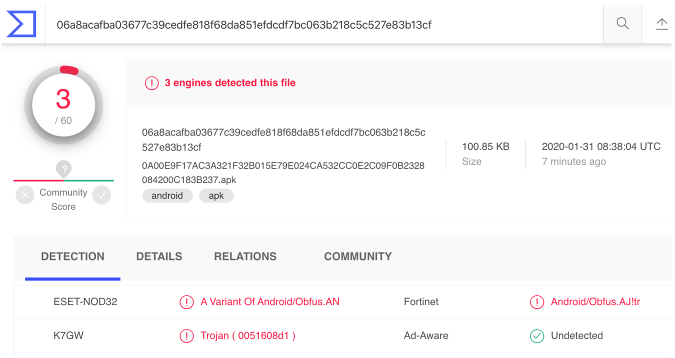
Deguard website and then uploads the piggybacked apps. Once the deobfuscation process has taken place, it automatically downloads the generated deobfuscated apps, which has been named with a suffix, i.e., the hash value of the original app. Because the Deguard tool is not open-source and its website does not support batch operation to handle a set of apps. Uploading each individual piggybacked app to its website is very slow and time consuming. Therefore, for each strategy, we randomly select 50 obfuscated apps to generate their deobfuscated apps, to make our evaluation as fair as possible.

B. Targeted Systems

To study the impact of code deobfuscation on detecting obfuscated piggybacked apps, we selected three types of Android malware detectors, including commercial anti-malware products, machine learning-based detectors, and similarity-based detectors.

1) Commercial Anti-malware Products: ***VirusTotal.*** As shown in Fig. 3, VirusTotal is a website (founded in 2004) which provides free suspicious file analysis services. Different from traditional anti-virus software, VirusTotal scans files through multiple commercially available anti-virus engines and then uses a variety of anti-virus engines to inspect the files uploaded to determine if the files are infected by viruses, worms, trojans or other types of malware. This greatly reduces the chances of the anti-virus software, either missing or not detecting a virus. Its detection rate is significantly better than when a single anti-virus product is used. Whilst no anti-virus software is 100% safe, the VirusTotal test results are more comprehensive and much more likely to be correct than if a single anti-virus engine was used. In addition, the compressed file structure (compressed file), file type - MD5, SHA1, SHA256 - can also be analyzed. In this study, after obfuscation and deobfuscation, if the number of anti-virus engines in VirusTotal that detect malware variants is greater than that of detecting original malware, the detection precision is considered to be increased, otherwise, the detection precision is considered to be decreased.

2) Machine Learning-Based Detectors: ***Drebin.*** Drebin is a lightweight method to detect Android malware [14]. It automatically infers the detection mode and directly identifies malware on mobile phones. First, Drebin collects as many features as possible from the application with static analysis.



06a8acafba03677c39cedfe818f68da851efdcdf7bc063b218c5c527e83b13cf

3 engines detected this file

3
/ 60

06a8acafba03677c39cedfe818f68da851efdcdf7bc063b218c5c527e83b13cf
0A00E9F17AC3A321F32B015E79E024CA532CC0E2C09F0B2328084200C183B237.apk

100.85 KB
Size

2020-01-31 08:38:04 UTC
7 minutes ago

ESET-NOD32	A Variant Of Android/Obfus.AN	Fortinet	Android/Obfus.AJ!tr
K7GW	Trojan (0051608d1)	Ad-Aware	Undetected

Fig. 3. Example of VirusTotal.

The collection of features mainly includes four sets of features from the manifest file (i.e., permissions, hardware components, APP components, and filtered intents) and four sets of features from disassembled code (i.e., restricted API calls, user permissions, suspicious API calls, and network addresses). These features are then organized into a set of strings, embedded in the vector space, and geometrically analyzed the patterns and combinations of these features geometrically analyzed. Finally, the embedded feature set is used to identify Android malware. Drebin learns a linear SVM classifier to discriminate between benign and malicious apps.

CSBD. CSBD is a method containing a python-based reimplementation of the Android malware detection [15]. CSBD uses control flow graph (CFG) signatures of methods in Android apps to detect malicious apps. Firstly, CSBD performs static analysis of the Android app's bytecode to extract a representation of the program's CFG, which is then expressed as character strings so that the similarity between Android apps can be established. This derived string representation of the CFG is an abstraction of the app's code, which retains information about the structure of the code but discards low-level details such as variable names or register numbers. Next, after having the abstract representation of an Android app's CFG, CSBD collects all basic blocks that compose and refer them as the features of the app. A basic block is taken as a sequence of instructions with only one entry point and one exit point, which represents the smallest piece of the app. By leveraging machine learning techniques and learning from the training dataset, the model exposes those basic blocks that statistically appear more frequently in Android malware.

3) Similarity-Based Detector: ***Androguard.*** Androguard is an open-source Android app analysis tool [16]. It implements a common similarity calculation method based on the original code at the method level. The feature extraction in Androguard is designed to generate an abstract representation of the method signature and statement. The latter represents the type from the statement (e.g., the if-statement, invoke statement) rather than the exact statement string. In addition, it also extracts constants, for example, numbers and strings as comparison features. Androguard then leverages normalized compression distance (NCD) to compute the similarity distance between two different methods. This is done by using state-of-the-art compressors to calculate the similarity between the two methods, rather than comparing all the statements in

TABLE IV
PRECISION OF DETECTING OBFUSCATED PIGGYBACKED APPS BY DIFFERENT DETECTORS

Type	Strategy	VirusTotal	Drebin	CSBD	Androguard	SimiDroid-C	SimiDroid-R	Average	STDEV
CFF	API_REFLECTION	54.7%	82.8%	37.6%	28.2%	100.0%	57.9%	60.2%	0.271
	API_INS	70.9%	77.5%	44.9%	35.7%	100.0%	58.6%	64.6%	0.233
IJC	BYTECODE	51.9%	33.3%	13.0%	61.8%	100.0%	38.9%	49.8%	0.297
IO	VARIABLE	88.5%	75.6%	81.5%	99.0%	100.0%	55.5%	83.3%	0.166
	PCM	77.3%	78.6%	82.0%	98.7%	3.6%	56.4%	66.1%	0.335
	BENIGN_CLASS	89.4%	72.3%	34.1%	58.9%	100.0%	57.0%	68.6%	0.238
	RESOURCE_IMAGE	64.2%	78.4%	73.0%	99.0%	100.0%	57.0%	78.6%	0.177
	RESOURCE_XML	76.2%	78.4%	77.9%	99.0%	100.0%	57.0%	81.4%	0.161
SO	STRING	101.7%	77.6%	51.6%	35.1%	100.0%	57.2%	70.5%	0.271
	BEN_PERMISSION	74.6%	81.0%	79.0%	99.0%	100.0%	57.0%	81.7%	0.161

Note. STDEV means the standard deviation of precision of each strategy to different detectors.

a given method.

SimiDroid. SimiDroid detects similar Android apps and explaining the identified similarities at different levels [17] (i.e., similarities at method level, component level, and resource level). In our study, as Androguard has compared the similarity of an app pair at the method level, we only use SimiDroid to compare APK file similarities at the component and resource levels.

We computed the similarity score of the given two apps (e.g., app1 and app2) using Equation 1. Given a predefined threshold, which can be computed based on a set of known piggybacked pairs, it is then possible to conclude with confidence that the given two apps are similar.

$$similarity = max(\frac{identical}{total - new}, \frac{identical}{total - deleted}), \quad (1)$$

where $identical$ denotes that when a given key/value entry is matched exactly the same in both maps; $similar$ is that when a given key/value entry slightly varies slightly from one app to the other in a pair and more specifically when the key is the same but values differ; new represents that when a given key/value entry exists only in map 2 but not in map 1; $deleted$ denotes that when a given entry existed in map 1, but is no longer found in map 2. In addition, $total$ = $identical$ + $similar$ + new + $deleted$.

C. Experiment Designs

We used the VirusTotal service to scan the generated obfuscated and deobfuscated piggybacked apps using anti-malware products by uploading the APK files to VirusTotal. For each uploaded app, VirusTotal returned a unique report. By analyzing the statistical results, we were able to obtain the detecting results of various commercial anti-virus products of both the obfuscated and deobfuscated piggybacked apps.

Additionally, we used two aforementioned machine learning-based detectors (i.e., Drebin and CSBD) to scan the obfuscated and deobfuscated apps. The two machine learning-based detectors were trained by the original dataset, i.e., 1,399 pairs of original/piggybacked apps. After the models were trained, they were then used to detect the generated obfuscated and deobfuscated datasets.

Furthermore, we utilized the two similarity-based detectors, including Androguard and SimiDroid, to detect the generated piggybacked apps. Androguard was used to compare original piggybacked malware at the method level and its corresponding malware that had been obfuscated by various strategies. In addition, SimiDroid was used at both the component level and the resource file level to compare the corresponding malware pairs.

The evaluation metric we used here are the precision of the malware detector. The precision is defined as $Precision = TP/(TP+FP)$, where TP is the number of correctly classified malware and FP is the number of benign apps which are predicted as malware. The reason why we choose precision as the evaluation metric is that if the detection precision of malware detectors is improved after deobfuscation, it means that the readability of the code is improved for the detectors.

IV. RESULTS AND ANALYSIS

Our empirical study aims to answer the following three research questions (RQs):

- **RQ1.** How is the precision of Android anti-malware detectors impacted by obfuscation strategies?
- **RQ2.** How is the precision of Android anti-malware detectors impacted by deobfuscation strategies?
- **RQ3.** How do different deobfuscation tools impact the precision of Android anti-malware detectors?

A. RQ1. Impact of Code Obfuscations

The precision of detecting obfuscated piggybacked apps by different detectors is shown in Table IV. Note that the precision of detecting original piggybacked apps (i.e., without code obfuscation) by each detector is 100%.

1) Commercial Anti-malware Products: ***VirusTotal.*** Table IV presents that if piggybacked apps are obfuscated, the detection precision of commercial anti-virus software will be greatly reduced. Of the 10 obfuscation strategies used, BYTECODE has the most severe impact on anti-virus products, since the precision of detection decreases the most. This is followed by API_REFLECTION, RESOURCE_IMAGE, and API_INS. The least severely affective obfuscation strategies on the anti-virus products of VirusTotal are VARIABLE and BENIGN_CLASS. Note that after being obfuscated by STRING, the detection precision of VirusTotal did not fall but slightly increase. STRING is to obfuscate string in the original code of an APK file, which can be speculated. Most anti-virus products already have a mature response for obfuscation strategy for strings, making them sensitive to string obfuscation.

TABLE V
PRECISION OF DETECTING DEOBFUSCATED PIGGYBACKED APPS BY DIFFERENT DETECTORS (SIMPLIFY)

Type	Strategy	VirusTotal	Drebin	CSBD	Androguard	SimiDroid-C	SimiDroid-R	Average	STDEV
CFF	API_REFLECTION	75.9% ↑	77.9% ↓	43.4% ↑	28.1% ↓	100.0% −	58.3% ↑	63.9% ↑	0.259
	API_INS	77.0% ↑	77.9% ↑	43.2% ↓	35.6% ↓	100.0% −	58.6% −	65.4% ↑	0.241
IJC	BYTECODE	59.0% ↑	63.3% ↑	33.0% ↑	61.8% −	100.0% −	38.9% −	59.3% ↑	0.235
IO	VARIABLE	75.8% ↓	76.3% ↑	69.4% ↓	99.9% ↑	100.0% −	55.5% −	79.5% ↓	0.175
	PCM	64.3% ↓	78.6% −	75.3% ↓	98.6% ↓	3.6% −	56.3% ↓	62.8% ↓	0.323
	BENIGN_CLASS	70.2% ↓	79.0% ↑	65.5% ↑	58.9% −	100.0% −	57.0% −	71.0% ↑	0.159
	RESOURCE_IMAGE	72.1% ↑	79.0% ↑	70.1% ↓	99.9% ↑	100.0% −	57.0% −	79.7% ↑	0.172
	RESOURCE_XML	73.2% ↓	79.0% ↑	74.5% ↓	99.9% ↑	100.0% −	57.0% −	80.6% ↓	0.167
SO	STRING	72.7% ↓	78.4% ↑	47.9% ↓	35.1% −	100.0% −	57.2% −	65.2% ↓	0.232
	BEN_PERMISSION	71.2% ↓	81.6% ↑	69.8% ↓	99.9% ↑	100.0% −	57.0% −	79.9% ↓	0.173

Note. "↑", "↓", and "−" after each cell value denote that the detection precision has been improved, decreased, or continued after deobfuscation compared to detecting obfuscated piggybacked apps. *STDEV* means the standard deviation of precision of each strategy to different detectors.

TABLE VI
PRECISION OF DETECTING DEOBFUSCATED PIGGYBACKED APPS BY DIFFERENT DETECTORS (DEGUARD)

Type	Strategy	VirusTotal	Drebin	CSBD	Androguard	SimiDroid-C	SimiDroid-R	Average	STDEV
CFF	API_REFLECTION	66.6% ↑	52.9% ↓	66.6% ↑	53.9% ↑	100.0% −	97.5% ↑	72.9% ↑	0.208
	API_INS	73.9% ↑	52.9% ↓	0.0% ↓	67.3% ↑	100.0% −	97.6% ↑	65.3% ↑	0.367
IJC	BYTECODE	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*	*NaN*
IO	VARIABLE	55.9% ↓	68.6% ↓	41.1% ↓	71.2% ↓	100.0% −	97.8% ↑	72.5% ↓	0.230
	PCM	46.8% ↓	74.5% ↓	46.6% ↓	68.7% ↓	100.0% ↑	98.2% ↑	72.4% ↑	0.235
	BENIGN_CLASS	49.7% ↓	50.9% ↓	25.5% ↓	64.7% ↑	100.0% −	98.2% ↑	64.8% ↓	0.293
	RESOURCE_IMAGE	54.9% ↓	60.7% ↓	36.0% ↓	71.2% ↓	100.0% −	97.8% ↑	70.1% ↓	0.250
	RESOURCE_XML	54.5% ↓	74.5% ↓	46.1% ↓	70.5% ↓	100.0% −	98.1% ↑	73.9% ↓	0.220
SO	STRING	88.9% ↓	60.7% ↓	12.0% ↓	89.1% ↑	100.0% −	97.6% ↑	74.7% ↑	0.337
	BEN_PERMISSION	50.2% ↓	80.3% ↓	53.1% ↓	70.6% ↓	100.0% −	98.2% ↑	75.5% ↓	0.214

Note. "↑", "↓", and "−" after each cell value denote that the detection precision has been improved, decreased, or continued after deobfuscation compared to detecting obfuscated piggybacked apps. For BYTECODE strategy, Deguard cannot generate deobfuscated application, so it is displayed as *NaN* in the table. *STDEV* means the standard deviation of precision of each strategy to different detectors.

2) Machine Learning-Based Detectors: ***Drebin.*** The obfuscation strategies also reduce the precision of the Drebin detector. However, apart from BYTECODE, the influences of other obfuscation strategies on the Drebin detector are similar. BYTECODE is a very successful evade strategy for this machine learning-based detector Drebin. BYTECODE changes the package name of Android piggybacked apps.

CSBD. Although CSBD is also a machine learning-based detector, it uses a completely different feature (i.e., CFG of the APK file) than Drebin. As shown in Table IV, the impacts of different obfuscation strategies for CSBD detector on obfuscated Android piggybacked apps for the CSBD detector are more pronounced than Drebin. In particular, STRING, BYTECODE, BENIGN_CLASS, API_INS, and API_REFLECTION significantly changes to CSBD's precision. However, the impact of other strategies on CSBD is similar to their impact on Drebin.

3) Similarity-Based Detector: ***Androguard.*** Androguard determines the similarity between two apps based on the method level. As shown in Table IV, the impact of different strategies on Androguard is significantly different from the other detectors used in this study. VARIABLE, PCM, RESOURCE_IMAGE, RESOURCE_XML, and BEN_PERMISSION have little effect on Androguard's detection performance, which means Androguard believes that the similarity score of the piggyback malware pair is close to one. Unlike the aforementioned strategies, STRING, BYTECODE, BENIGN_CLASS, API_INS, and API_REFLECTION have a huge impact on the performance of the Androguard. This is because these strategies are mainly changed at the method level, which is sensitive to the Androguard.

SimiDroid-C. The similarity between the two apps is determined by comparing the component level methods. It can be observed from Table IV that SimiDroid's resilience is excellent against all the obfuscation strategies except PCM. This is because PCM is to obfuscate the name of the component in the source code.

SimiDroid-R. The comparison of resources is used to determine the similarity between the two apps' APK files. Table IV shows that all the obfuscation strategies drop the detection performance of this detector. Most obfuscation strategies will drop their performance from 100% to 55%, while BYTECODE falls even more to 40%.

Finding #1: *The detection precision of all the obfuscating strategies examined can decrease from around 20% to 50% of the detection precision of all the detectors examined. In particular, BYTECODE has the most significant impact on the detection precision, while BEN_PERMISSION has the least impact.*

B. RQ2. Impact of Code Deobfuscations

Table V and Table VI show the precision of detecting deobfuscated piggybacked apps by different detectors. Note that the values in the row of BYTECODE of Table VI are *NaN*, since Deguard can not deobfuscate obfuscated apps using BYTECODE strategy.

1) Commercial Anti-malware Products: ***VirusTotal.*** Table V shows that the detection precision after deobfuscation is worse than that of detecting the obfuscated piggybacked apps by six strategies (i.e., VARIABLE, PCM, BENIGN_CLASS, RESOURCE_XML, STRING, and BNE_PERMISSION). Identifier-renaming strategies (e.g., STRING, VARIABLE, PCM, and BENIGN_CLASS) can decrease the detection precision after deobfuscations. These strategies change the previous names without knowledge of the original names, so the renaming deobfuscation is very difficult. However, deobfuscators can convert invisible characters into visible characters or long strings to short strings. As a result, a simple string replacement makes deobfuscated piggybacked apps look less suspicious, making them more likely to evade malware detectors.

On the contrary, four strategies (i.e., BYTECODE, RESOURCE_IMAGE, API_INS, and API_REFLECTION) by using Simplify can increase the precision of VirusTotal for detecting deobfuscated piggybacked apps by 7.2%, 7.8%, 6.1%, and 21.2%, respectively, compared to detecting obfuscated apps. Because obfuscated Android piggybacked apps after deobfuscation with these strategies can restore its malicious behaviour, thereby increasing the detection precision.

Moreover, for the deobfuscation tool Deguard (Table VI), only two deobfuscation strategies (i.e., API_INS and API_REFLECTION) can increase the precision of detecting deobfuscated piggybacked apps. The rest of strategies make the detection precision worse than that of detecting obfuscated malware.

Finding #2: *The precision of commercial anti-malware products for detecting deobfuscated piggybacked apps can be improved by engaging CFF-related strategies (i.e., API_REFLECTION and API_INS) for both Simplify and Deguard. However, after deobfuscations with IO-related (except RESOURCE_IMAGE for Simplify) and SO-related strategies, the precision of detecting debofuscated piggybacked apps would be decreased.*

2) Machine Learning-Based Detectors: ***Drebin.*** After deobfuscation tool (i.e., Simplify) using all the strategies (except API_REFLECTION and PCM), the precision of detecting deobfuscated piggybacked apps can be slightly improved compared to detecting obfuscated malware. In particular, BYTECODE strategy has a significant impact on the precision of Drebin for detecting deobfuscated piggybacked apps, i.e., the detection precision has increased from 33.3% to more than 63.3%. However, after deobfuscation using Deguard, the precision would be decreased.

Drebin's mechanism is to collect a large amount of information in the APK as features. As a result, slight modifications of the code in the APK file, such as simple renaming, are not very sensitive to Drebin. Therefore, the more features are taken for a machine learning-based detector, the better the detection precision and obfuscation resilience capability.

Finding #3: *Simplify helps improve the precision of Drebin on detecting piggybacked apps obfuscated with all the strategies except API_REFLECTION and PCM, while deobfuscations using Deguard decreases the detection precision.*

CSBD. For Simplify, BYTECODE, BENIGN_CLASS, and API_REFLECTION can significantly improve the precision of CSBD for detecting deobfuscated piggybacked apps, compared to when used on obfuscated piggybacked apps. In addition, for Deguard, only API_REFLECTION strategy can improve the detection precision. For all other strategies, the detection results are worse than that of detecting obfuscated piggybacked apps.

Finding #4: *Simplify with BYTECODE, BENIGN_CLASS, and API_REFLECTION helps improve the precision of CSBD when detecting deobfuscated piggybacked apps. For Deguard, only API-related strategy (API_REFLECTION) improves the precision of CSBD, while the rest strategies decrease the precision.*

3) Similarity-Based Detector: ***Androguard.*** For the deobfuscation tool Simplify, the precision of detecting deobfuscated piggybacked apps is similar to the detection of obfuscated piggybacked apps. Therefore, we can conclude that code deobfuscation has no significant impact on the similarity-based detector, which uses the method level feature. However, for Deguard, CFF-related strategies (i.e., API_REFLECTION and API_INS) significantly improve the precision of detecting deobfuscated malware compared with detecting obfuscated malware.

Finding #5: *After deobfuscation using Simplify with all strategies, the precision of Androguard for detecting deobfuscated piggybakced malware is similar to detecting obfuscated piggybacked apps. However, Deobfuscation using Deguard with CFF-related strategies (i.e., API_REFLECTION and API_INS) can significantly improve the detection precision of Androguard for detecting deobfuscated piggybacked apps.*

SimiDroid-C. For Simplify, except PCM, the resilience of SimiDroid is excellent against almost all the deobfuscation strategies. Similar to the method level similarity-based detector, the deobfuscation strategy has little effect on the component level similarity-based detector. Moreover, the detection precision of the similarity-based detector is not affected by Deguard's deobfuscation, especially PCM, which is strongly influenced by Simplify deobfuscation.

SimiDroid-R. With deobfuscation using Simplify, there is a slight change in the detection precision. However, using Deguard for deobfuscation, the precision of SimiDroid-R for detecting deobfuscated malware is greatly improved when using Deguard with all the strategies.

Finding #6: *Code deobfuscation has no significant impact on a similarity-based malware detector using a component-level feature to detect deobfuscated piggybacked apps. In addition, a deobfuscation tool using Deguard can significantly improve the precision of a similarity-based malware detector using a resource-level feature to detect deobfuscated malware.*

By analyzing the experimental results, after deobfuscation, it can be found that many detectors' precision decreased for obfuscated malware under different strategies. The reason is that, in many scenarios, the deobfuscators can not reinstate the obfuscated code back to the original code, but adding additional 'noise' (the modified code or the newly introduced code fragments by the deobfuscators), therefore, the results of the malware detectors can be negatively impacted.

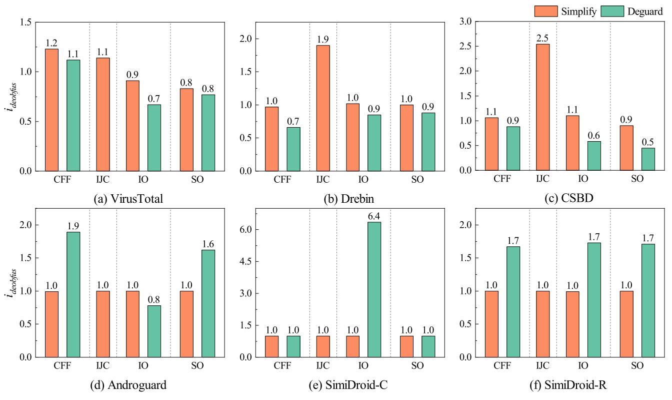


Fig. 4. The impact of code deobfuscation tools on different detectors. (a) VirusTotal, (b) Drebin, (c) CSBD, (d) Androguard, (e) SimiDroid-C, and (f) SimiDroid-R.

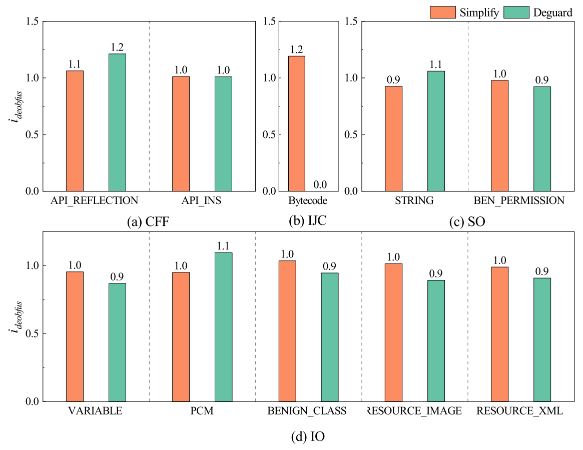


Fig. 5. The impact of code deobfuscation tools on different types of obfuscation strategies. (a) CFF, (b) IJC, (c) SO, and (d) IO.

C. RQ3. Impact of Deobfuscation Tools

To compare the impact of deobfuscation on the detection precision using different deobfuscation tools (i.e., Simplify and Deguard), we defined a parameter $i_{deobfus}$ as follows.

$$i_{deobfus} = \frac{acc_{deobfus}}{acc_{obfus}} \tag{2}$$

where $acc_{deobfus}$ denotes the precision of detecting deobfuscated piggybacked apps, as shown in Tables V and VI, while acc_{obfus} is the precision of detecting obfuscated piggybacked apps, as depicted in Table IV. Note that If $i_{deobfus}$ is larger than 1, it means the detection precision has increased after deobfuscation. Otherwise, it means the detection precision has decreased after deobfuscation.

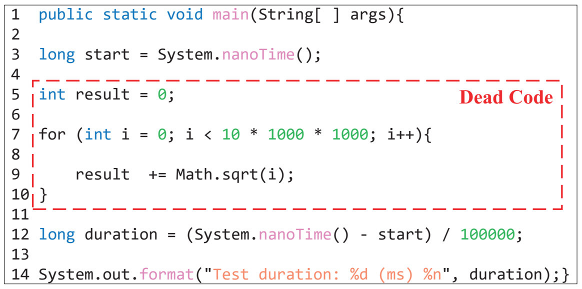
1) Impact on Detectors: Fig. 4 shows the impact of code deobfuscation tools on different detectors. For each detector, we compare the impact of each strategy type using different code deobfuscation tools. Note that each column in the figure is calculated by averaging the $i_{deobfus}$ belonging to the corresponding type of strategy and detector.

We can observe from Fig. 4(a), (b), and (c) that Simplify has a greater impact on commercial anti-malware products (i.e., VirusTotal) and machine learning-based detectors (i.e., Drebin and CSBD) than Deguard. For example, after Simplify deobfuscation with the strategy types of CFF and IO, the detection precision of CSBD has greatly improved, compared with the deobfuscation using Deguard.

Deguard has a greater impact on similarity-based detectors (i.e., Androguard and SimiDroid) than Simplify, as shown in Fig. 4(d), (e) and (f). For example, after deobfuscation using Deguard under the strategy types of CFF, IO, and SO, the detection precision of SimiDroid-R has greatly improved, compared with the deobfuscation using Simplify.

Finding #7: *Different code deobfuscation tools have a different impact on anti-malware detectors. In particular, by using Simplify for commercial anti-malware products (i.e., VirusTotal) and machine learning-based detectors (i.e., Drebin and CSBD) the improvement in the precision of detecting deobfuscated piggybacked apps is greater. However, for similarity-based detectors (i.e., Androguard and Simidroid), Deguard performs better than Simplify.*

2) Impact on Obfuscation Strategies: Fig. 5 shows the impact of code deobfuscation tools on different types of strategies. Note that each column in the figure is calculated by averaging the $i_{deobfus}$ belonging to the same strategy and corresponding to the same type. From Fig. 5, we can not conclude that the code deobfuscation tools have any significantly different impact on each strategy in the same type, when detecting deobfuscated piggybacked apps after code deobfuscation.


```
public static void main(String[ ] args){

long start = System.nanoTime();

int result = 0;                                   Dead Code

for (int i = 0; i < 10 * 1000 * 1000; i++){

    result  += Math.sqrt(i);

}

long duration = (System.nanoTime() - start) / 100000;

System.out.format("Test duration: %d (ms) %n", duration);}
```

Fig. 6. Case study: insert junkcode.

V. Case Studies

This section presents case studies to demonstrate representative deobfuscated piggybacked apps using different deobfuscation tools.

Insert Junk Code (IJC). Malware ID-FFCE7D9B[1] is deobfuscated by Simplify using an insert junk code (IJC) obfuscation strategy. This strategy involves obfuscators inserting extra code snippets into the script, including some variables and functions that are never referenced or called after they are fixed, as shown in Fig. 6. This code may be executed, but it will not affect the overall execution result of the script. The ideal result would be that deobfuscators remove it from the code. However, the Simplify does not have the ability to discern and delete the IJC. Therefore, the code structure of this obfuscated malware can not be restored to the original app. Consequently, the deobfuscation does not contribute to more accurate detection of this malware. This malware eventually evade both the machine learning and the similarity-based

[1]Since each malware has a unique hash value (MD5), we used the first eight digits to denote its hash values.

```
private void CalculatePayroll(SpecialList a){
    while (a.HasMore()){
        b = a.GetNext(true);
        b.UpdateSalary();
        DistributeCheck(b);
    }
}
```

(a) Obfuscated Code

```
private void CalculatePayroll(SpecialList DBhelper){
    while (DBhelper.HasMore()){
        DB = DBhelper.GetNext(true);
        DB.UpdateSalary();
        DistributeCheck(DB);
    }
}
```

(b) Deobfuscated Code

Fig. 7. Case study: identifier renaming.

```
public class Foo{
    private String encrypted = "ÎÒ´òµÄ¾ÍÊÇÂÒÂëれ腥地BBSΘ航BBSい锣更";
    private String key = "ÂBÊÇBれëSÂ地Í腥ΘÒ";
    private String mySecret = MyDecrytUtil.decrypt(encrypted, key);
}
```

(a) Obfuscated Code

```
public class Foo{
    private String mySecret = "http://textspeier.de";
}
```

(b) Deobfuscated Code

Fig. 8. Case study: string obfuscation.

detectors. VirusTotal's detection rate has decreased compared to the obfuscated one. This is because the newly generated junk code introduces additional noise which adversely affected the performance of the anti-virus vendors.

Identifier Obfuscation (IO). Malware ID-E4A8A133 is deobfuscated by Simplify for identifier obfuscation (IO), which is the most common obfuscation technique. For example, a class/method/variable name is often obfuscated into an arbitrary identifier composed of both uppercase and lowercase letters and numbers, which can be used to evade machine-learning-based detectors which rely on recognizing and matching identifier. However, the similarity-based detector can detect this malware. Although Simplify can replace some obfuscated with more meaningful names (e.g., an example is illustrated in Fig. 7), the deobfuscation is still unable to restore the app back to the original malware, thus also evading the machine-learning-based detectors.

String Obfuscation (SO). Malware ID-A9D69887 is deobfuscated by Deguard against string obfuscation. Fig. 8 indicates that decryption class generates a key through the hash code value of the APK certificate and the internal key reference array (ref_key), and reads and decrypts the original encrypted DEX file. Unlike using AES for the string encryption option, this class encryption option uses its own decryption algorithm. For this malware, after deobfuscation, the string is decrypted to be a fraud e-commerce website, thus, many more vendors in VirtualTotal are able to detect this malware after deobfuscation.

VI. Related Work

A. Android Malware Detection

There are several existing solutions to Android malware detection [14, 18–22], most of which give a binary decision to identify whether or not an app is malicious. Arp *et al.* [14] proposes the Drebin, through a two-class SVM by performing a lightweight static analysis to extract API calls and manifest files as the input features. MaMaDroid [23] leveraged sequences of abstracted method calls to create a probabilistic representation of program behaviors in the form of Markov chains. Aafer *et al.* [18] mined API features for malware detection in Android using a lightweight KNN-based binary classification. Hou *et al.* [24] proposed an Android malware detection approach using a heterogeneous information network. Kim *et al.* [25] presented a deep-learning-based approach to malware detection by utilizing the features extracted from an Android app.

Code-clone-based repackaging application detection is one of the most common code similarity detection methods and is widely used in Android repackaged application detection. Zhou *et al.* [26] used a fuzzy hashing technique on the opcode to generate fingerprint information representing the application. It then uses the edit distance to calculate the similarity of the two applications. Hanna *et al.* [27] used k-gram to process the application's opcode and a Bloom-filter based feature hash algorithm to generate the vector representation of the application. The representation was then compared using the Jaccard similarity distance to calculate the similarity of the two applications. The two methods in [26] and [27] can achieve large-scale application similarity comparison, but if IJC is inserted or adjusted in the source code, this strategy is ineffective. Zheng *et al.* [28] used the program dependency graph (PDG) as the feature representation of the application. A PDG is generated for each method of each class in the application, and similarity matching is used to detect similar applications.

B. Android Obfuscation Strategies

Some work on Android-specific obfuscation tools has previously been conducted. For example, Rastogi *et al.* [6] presented Droid Chameleon, a tool for obfuscating Android apps. Zheng *et al.* [29] proposed ADAM, a framework for obfuscating Android apps and testing them on anti-malware products. Ghosh *et al.* [30] used a reflection call mechanism in Android application development to hide calls to sensitive APIs and access to sensitive data. But this method can be identified by inspecting whether Java reflection calls are being used. Harrison *et at.* [31] analyzes the code obfuscation technology that resists reverse engineering of Android applications. With this technology, obfuscation is divided into name obfuscation, control flow obfuscation, and instruction encoding obfuscation. Naming obfuscation converts the name of a package, class, or method in the code into a meaningless string. However, since DEX bytecode is always stored in memory in the form of clear text during execution, an attacker can dynamically analyze the name obfuscation and implement the DEX bytecode loading logic [32]. There are also some other studies also focused on the effects of obfuscations on anti-malware products, without proposing new obfuscation tools. Pomilia [33] studied the performance of nine anti-malware products on an obfuscated. Maiorca *et al.* [34] studied the effects of code obfuscated by a single tool on 13 anti-malware products.

C. Android Deobfuscation

Schulz *et al.* [35] proposed a deobfuscation method for the string encryption option provided by DexGuard, an obfuscation tool for Android. DexGuard first extracts the decoded string, and later reverse obfuscation is performed by replacing the encrypted string with the extracted decryption string and deleting the decryption method. However, if multiple class encryption options are applied so that the decryption method cannot be found, there is the limitation that reverse obfuscation cannot be performed. Piao *et al.* [36] also analyzed DexGuard and analyzed string encryption and class encryption options. In the case of the string encryption option, it was shown that the decrypted string is output to the log when the decryption method is called and that the log can be extracted by demonstrating that the log is identical to the original string. The method, key, and initial vector are extracted, and the decrypted class file is extracted through the decryption method call. In the case of APKs, the approach cannot perform deobfuscation of strings and cannot extract strings of routines that are not executed. If the string such as the attacker's IP address or the name of the malicious API is called in and the malicious APK is not deobfuscated, it may not be able to determine whether the APK is malicious.

VII. Conclusion

This paper presents a large-scale empirical study of code deobfuscations on detecting obfuscated Android piggybacked apps. First, we generated obfuscated piggybacked apps by using 10 obfuscation strategies on 1,399 Android piggybacked apps. Using the obfuscated apps generated, we then used two commonly used deobfuscation tools, i.e., Simplify and Deguard, to produce deobfuscated piggybacked apps. The total number of generated obfuscated and deobfuscated piggybacked apps was 11,378. Next, we have conducted an empirical evaluation of the generated obfuscated and deobfuscated piggybacked apps on three types of Android anti-malware detectors, including commercial anti-malware products (VirusTotal), machine learning-based detectors (Drebin and CSBD), and similarity-based detectors (Androguard and SimiDroid). The analysis was performed on two aspects: the impact of code obfuscations and deobfuscations. Along with seven findings, our results provided useful insights into the design of Android malware detectors against obfuscations and deobfuscations.

VIII. Acknowledgement

This work is supported in part by the Australian Research Council under Grant DP200101328 (by Ivor W. Tsang and Yulei Sui), in part by the National Natural Science Foundation of China under Grants 62002163 and 61772055 (by Guanping Xiao and Zheng Zheng).

References

[1] T. I. Murphy, "Android-statistics & facts," https://www.statista.com/topics/876/android, 2018.

[2] Y. Tang, Y. Sui, H. Wang, X. Luo, H. Zhou, and Z. Xu, "All your app links are belong to us: Understanding the threats of instant apps based attacks," *ACM SIGSOFT FSE*, 2020.

[3] "Apktool," https://ibotpeaches.github.io/Apktool/, 2010, last accessed March 2019.

[4] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE S&P*. IEEE, 2012, pp. 95–109.

[5] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013.

[6] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *TIFS*, vol. 9, no. 1, pp. 99–108, 2013.

[7] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *ICSE*, 2018.

[8] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android hiv: A study of repackaging malware for evading machine-learning detection," *TIFS*, 2019.

[9] C. Fenton, "Simplify," https://github.com/CalebFenton/simplify, 2016.

[10] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *CCS*, 2016.

[11] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *TIFS*, 2017.

[12] R. Baumann, M. Protsenko, and T. Müller, "Anti-proguard: Towards automated deobfuscation of android apps," in *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, 2017, pp. 7–12.

[13] Wikipedia contributors, "Selenium (software) — Wikipedia, the free encyclopedia," 2020, [Online; accessed 1-February-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Selenium_(software)&oldid=937169086

[14] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, 2014.

[15] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, Y. Le Traon *et al.*, "Empirical assessment of machine learning-based malware detectors for android," *Empirical Software Engineering*, 2016.

[16] A. Desnos *et al.*, "Androguard-reverse engineering, malware and goodware analysis of android applications," *URL code. google. com/p/androguard*, 2013.

[17] L. Li, T. F. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017.

[18] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *SecureComm*, 2013.

[19] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale." in *USENIX*, vol. 15, 2015.

[20] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *FSE*, 2014.

[21] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *ICSE*, 2015.

[22] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, "Familial clustering for weakly-labeled android malware using hybrid representation learning," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3401–3414, 2019.

[23] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[24] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Hindroid: An intelligent android malware detection system based on structured heterogeneous information network," in *SIGKDD*, 2017.

[25] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *TIFS*, vol. 14, no. 3, pp. 773–788, 2019.

[26] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *CODASPY*, 2012.

[27] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *DIVMA*, 2012.

[28] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *ESORICS*, 2012.

[29] M. Zheng, P. P. Lee, and J. C. Lui, "Adam: an automatic and extensible platform to stress test android anti-virus systems," in *DIMVA*. Springer, 2012, pp. 82–101.

[30] S. Ghosh, S. Tandan, and K. Lahre, "Shielding android application against reverse engineering," *International Journal of Engineering Research & Technology*, vol. 2, no. 6, pp. 2635–2643, 2013.

[31] R. Harrison, "Investigating the effectiveness of obfuscation against android application reverse engineering," *Royal Holloway University of London, Tech. Rep. RHUL-MA-2015-7*, 2015.

[32] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *ESORICS*. Springer, 2015, pp. 293–311.

[33] M. Pomilia, "A study on obfuscation techniques for android malware," *Sapienza University of Rome*, p. 81, 2016.

[34] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[35] H. Schulz, D. Titze, J. Schutte, T. Kittel, and C. Eckert, "Automated deobfuscation of android bytecode," *Department of Computer Science, The University of Munchen, Germany, July*, 2014.

[36] Y. Piao, J.-H. Jung, and J. H. Yi, "Server-based code obfuscation scheme for apk tamper detection," *Security and Communication Networks*, 2016.