A New Technique for Detecting Android App Clones Using Implicit Intent and Method Information

Byoungchul Kim
Dept. of Software Science
Dankook University
Gyeonggi-do, Republic of Korea
gurukbc@dankook.ac.kr

Soyeon Jeon
Dept. of Software Science
Dankook University
Gyeonggi-do, Republic of Korea
polkmn35@dankook.ac.kr

Jaemin Jung
Dept. of Computer Science and Engineering
Dankook University
Gyeonggi-do, Republic of Korea
snorlax@dankook.ac.kr

Seong-je Cho
Dept. of Computer Science and Engineering
Dankook University
Gyeonggi-do, Republic of Korea
sjcho@dankook.ac.kr

Sangchul Han
Dept. of Software Technology
Konkuk University
Chungcheongbuk-do, Republic of Korea
schan@kku.ac.kr

Jongmoo Choi
Dept. of Computer Science and Engineering
Dankook University
Gyeonggi-do, Republic of Korea
choijm@dankook.ac.kr

Abstract — Detecting repackaged apps is one of the important issues in the Android ecosystem. Many attackers usually reverse engineer a legitimate app, modify or embed malicious codes into the app, repackage and distribute it in the online markets. They also employ code obfuscation techniques to hide app cloning or repackaging. In this paper, we propose a new technique for detecting repackaged Android apps, which is robust to code obfuscation. The technique analyzes the similarity of Android apps based on the method call information of component classes that receive implicit intents. We developed a tool *Calldroid* that implemented the proposed technique, and evaluated it on apps transformed using well-known obfuscators. The evaluation results showed that the proposed technique can effectively detect repackaged apps.

Keywords—Andorid app; clone detection; code obfuscation; intent; class method

I. Introduction

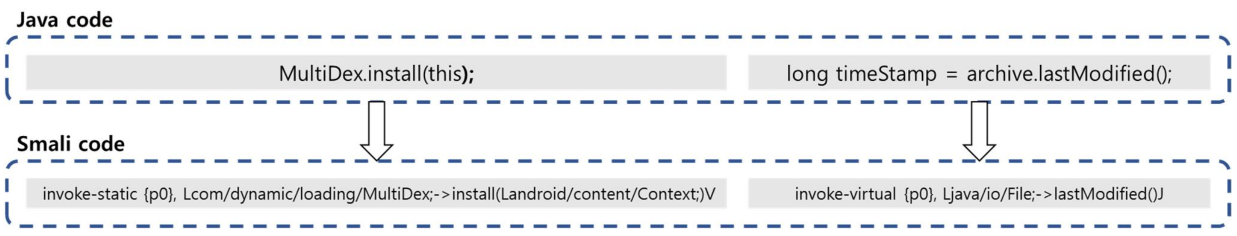
Detecting repackaged apps is one of the most important research issues to be addressed. Cloned apps and repackaged malware significantly affect the Android ecosystem [1]. According to [2], 86% of Android malware make use of the repackaging technique to disassemble or reverse engineer a legitimate app, modify its logics or add malicious code, and package it back. Then the repackaged app is distributed in the online markets. Attackers even crack or insert malicious logics into apps to leak users' private information, to bypass in-app payment procedure, to replace developers' advertisement IDs, to intercept SMS or to make a call without users' notice. Consequently, to secure the Android ecosystem, apps distributed in the online markets should be scanned to filter out cloned or repackaged apps.

There have been many researches on detecting Android app clones based on app birthmarks [3, 4, 5, 6, 7, 8]. A birthmark is an intrinsic characteristic of an app. Many app features such as Dalvik bytecodes, API/method-level information, class-level information and UI/resource information can be a birthmark of app. If the birthmarks of two apps are similar to each other, it is adjudged that one of them is cloned or repackaged.

On the other hand, attackers employ code obfuscation techniques to bypass cloned app detection that is based on the similarity analysis of app birthmarks. Code obfuscation is a technique that transforms a program's source codes or execution codes to make it difficult to understand. App developers apply code obfuscation to protect their codes from tampering or reverse engineering, while attackers leverage code obfuscation to hide the logics of malicious codes or to change app features so that a cloned app's birthmark may be different from its original one. Proguard [9], DexGuard [10], DashO [11] are well-known obfuscators for Android apps.

Software birthmarks should be robust to code obfuscation. Many existing studies use disassembled Dalvik bytecode and hashing approaches to detect cloned apps and repackaged malware. Since this approach is sensitive to the instruction sequences of disassembled codes, they may not detect cloned or repackaged apps if code obfuscation is applied [1]. Other approach including DNADroid [4] utilizes a program dependency graph extracted from disassembled codes. This approach is not scalable since comparing program dependency graph is computation intensive.

In this paper, we propose a repackaged app detection technique that is scalable and robust to obfuscation techniques



```
MultiDex.install(this);
long timeStamp = archive.lastModified();
```

```
invoke-static {p0}, Lcom/dynamic/loading/MultiDex;->install(Landroid/content/Context;)V
invoke-virtual {p0}, Ljava/io/File;->lastModified()J
```

Fig. 1. Example of `invoke-*` instructions in smali code

including code obfuscation. This technique extracts the birthmark from the classes that receives implicit intents, which is a kind of message between Android apps. Since most obfuscators do not obfuscate the classes that receives implicit intents, the features extracted from those classes are not modified during obfuscation. We develop and evaluate a cloned app detection tool *Calldroid* that implements the proposed technique. We collect Android apps from F-droid [12] and obfuscate them using ProGuard, DexGuard and DashO, then measure the similarity between the original app and the obfuscated one. We also compare the performance of Calldroid with a similarity comparison framework SimiDroid [3].

II. Background

A. Code Obfuscation

ProGuard, DexGuard, DashO are well-known Android obfuscation tools. The goal of these tools is to make reverse engineering difficult and secure the codes and resources of apps. Table 1 shows the obfuscation techniques the tools support. The renaming technique changes identifiers such as package name, class name and method name so that analyzers cannot understand the source codes easily. String encryption technique encodes string literals using an encryption algorithm or cryptographic cipher and makes them unreadable. Usually the encoded strings are decoded at runtime by passing them to a decoding function. Control-flow obfuscation hampers the comprehension of the logical flow of a program by inserting linked jump instructions, dead code, dummy method, etc.

TABLE I. Well-known Android obfuscators

	ProGuard	**DexGuard**	**DashO**
Package renaming	support	support	support
Class renaming	support	support	support
Method renaming	support	support	support
String encryption	not support	support	support
Control-flow obfuscation	support	support	support

B. Intent

Android apps have an XML document AndroidManifest.xml in the app's root directory. AndroidManifest.xml declares important information such as the description of components of the app; the classes from which the app starts execution, the permissions used by the app, the actions to activate a specific component, etc. The manifest file also contains the information on intents, which is a kind of messages. Android intent system is used for inter- and intra-app communication. There are two types of intents; explicit intent and implicit intent [13, 14]. An explicit intent specifies its recipient by supplying the exact class name of the component. Thus, explicit intents are mainly used for intra-app communication. While, an implicit intent just specifies an action to be performed to the system. The actual recipient is determined by the system. This procedure is called intent resolving. In AndroidManifest.xml, all components (i.e., Activity, Receiver, Service) can declare what kind of intents it can respond to using <intent-filter> tag. The system matches an implicit intent against intent filters declared by installed apps, and discovers a target component.

The manifest file declares the type and attributes of each component class of the app. A component class that is to receive explicit intents from other apps declares an attribute "android:exported=true". A component class that is to receive implicit intents declares intent filters using <intent-filter> tag, and the default value of attribute android:exported is true so that components of other apps can see this component. If a developer wants to hide a component class that receives implicit intents from other apps, he or she declared the attribute "android:exported=false" explicitly.

If a component class receives intents only within its app or does not receive intents from other apps, obfuscating the component class does not affect the intra-app communication and the functionality of the component. On the other hand, if a component class receives intents from other apps, the information on the component class should be clearly exported to others. The component classes that declare intent filters in AndroidManifest.xml, i.e., that receive implicit intents, are exported by default. To provide smooth inter-app communication and the functionality of exported components, much information of them is not changed by obfuscators. Specifically, DashO obfuscates only the classes that are not referenced by other apps [11]. Proguard does not obfuscate all the components declared in AndroidManifest.xml by default [9].

C. Smali code and Method invocation

Smali code is an assembly language for the Dalvik bytecode. Its format is very simple and convenient for static analysis. We decompile APKs and obtain smali codes, then analyze the codes to produce the information on method invocation. In the smali code, method invocations always start with invoke-* or execute-* instructions. Table 2 lists various

TABLE II. INVOKE INSTRUCTIONS IN SMALI CODE

Mnemonic	Description
invoke-virtual	Invokes a virtual method
invoke-super	Invokes the virtual method of the immediate parent class.
invoke-direct	Invokes an instance method
invoke-static	Invokes a static method
invoke-interface	Invokes an interface method

invoke-* instructions. The kind of the invoked method determines a specific instruction; invoke-virtual, invoke-super, etc. As shown in Fig. 1, invoke-* instructions are followed by the class name, method name and arguments of the invoked method. We use the invoke-* instructions as the main feature for app birthmark.

III. APPROACH

In this study, we extract the information on class and method invocations for app features from component classes that receive implicit intents, and compare the similarity between Android apps. As explained in Section 2.B, component classes that receive intents are involved in the communication between apps. Hence, most obfuscators do not obfuscate such classes.

A. Feature Extraciton

By decompiling Android executable file (APK file), we can obtain the app's smali codes and AndroidManifest.xml. We identify component classes containing <intent-filter> tag, search for a smali file of which name matches the component class name, and extract features from its smali file. Specifically, we search for method invocation codes from a smali file. As explained in Section 2.C, those codes start with "invoke-*". Fig. 1 shows an example of java-to-smali code conversion. A method invocation code contains the fully qualified name (including package and class name) of the invoked method and its parameters. Fig. 2 illustrates this process.

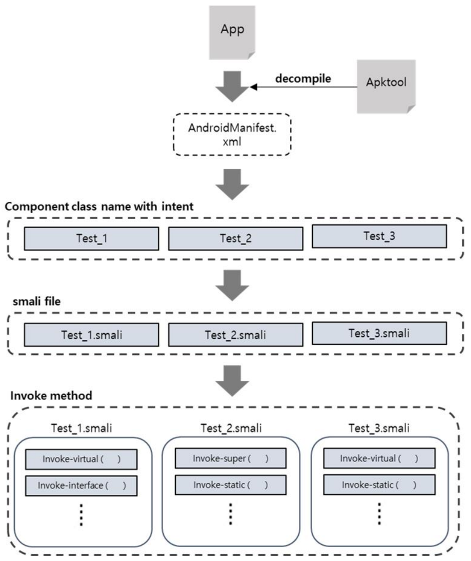


Fig. 2. Process of extracting feature information

We extract all method invocation features from each component class that declares intent filters and construct a birthmark of an app. A birthmark is defined as $BM(app) = \{C(c_1), C(c_2), \ldots, C(c_n)\}$ where c_i is a component class declaring intent filters and $C(c)$ is a collection of method invocation features extracted from component class c. $C(c)$ is defined as $($*component-name*$, m_1, m_2, \ldots, m_k)$ where k is the number of method invocations, and m_i is defined as (*fully-qualified-method-name*, *parameter-type*$_1$, *parameter-type*$_2$, …).

B. Measuring Similarity

To measure the similarity of birthmarks of two apps, we count the matched pair of method invocation. Two method invocations are said to be matched if the following three conditions are met; (1) their component names are identical (2) their fully qualified method names are identical (3) their lists of parameter types are identical. For example, in Fig. 3, the second method invocation of MainActivity of App1 and App2 are matched. However, the first method invocation of

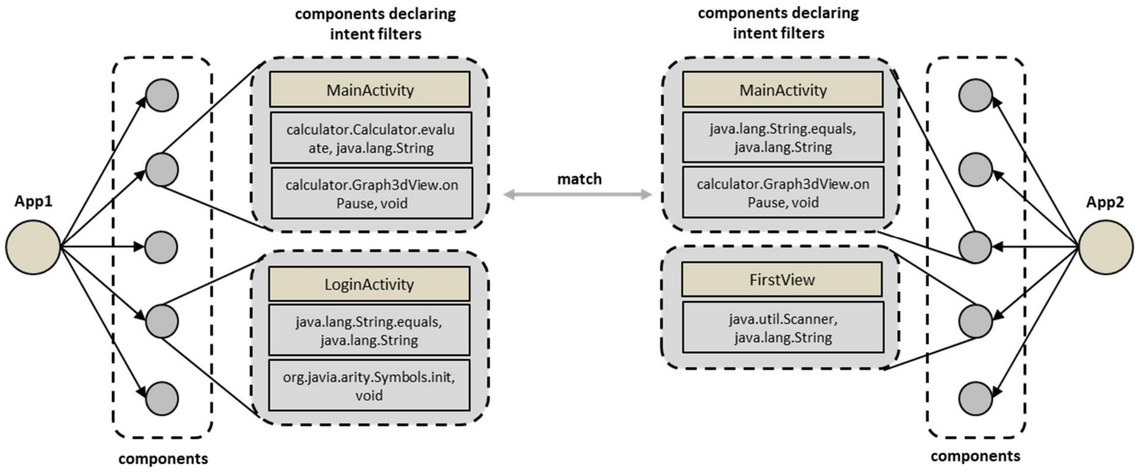

components declaring intent filters
MainActivity
calculator.Calculator.evaluate, java.lang.String
calculator.Graph3dView.onPause, void
LoginActivity
java.lang.String.equals, java.lang.String
org.javia.arity.Symbols.init, void
App1
components
match
components declaring intent filters
MainActivity
java.lang.String.equals, java.lang.String
calculator.Graph3dView.onPause, void
FirstView
java.util.Scanner, java.lang.String
App2
components

Fig. 3. Similarity measurement in CallDroid

LoginActivity of App1 and the first method invocation of MainActivity of App2 are not matched because their component names are not identical.

The birthmark similarity of two apps is defined as the following equation.

$$S(A, B) = \frac{match(A, B)}{count(A) + count(B) - match(A, B)}$$

where *match*(*A,B*) is the number of matched pair of method invocation and *count*(*app*) is the total number of method invocation of an app. For example, *match*(*App1, App2*)=1, *count*(*App1*)=4 and *count*(*App2*)=3 in Fig. 3. The birthmark similarity between App1 and App2 is *S*(*App1, App2*) = 1 / (4+3−1) = 0.167.

IV. Evaluation

To assess the effectiveness of CallDroid, we conduct a real implementation-based experiment. We generate a testing dataset that consists of Android apps, gathered from the F-droid [12] and repackaged using the three popular obfuscation tools, namely ProGuard [9], DexGuard [10] and DashO [11]. Details of the number of generated apps and obfuscation options are given in Table 3. We apply diverse options including optimization, renaming, string encryption and control flow modification to evaluate the capability of CallDroid on a variety of hiding techniques.

In this section, we first describe how sensitive CallDroid is on the threshold value for measuring similarity. Then, we quantitatively compare CallDroid with a well-known app cloning detection tool, SimiDroid [3]. Finally, we discuss the performance of CallDroid on different obfuscation options.

TABLE III. Details of testing apps: number and obfuscation options

	Transformation method (Obfuscator's option)	Number of obfuscated apps	
ProGuard	Default	92	184
	Optimize	92	
DexGuard	Renaming	92	184
	String encryption	92	
DashO	Renaming	126	252
	Control flow modification	126	
Total		620	

A. Threshold analysis

One important control parameter for CallDroid is the similarity threshold. When the similarity expressed in Section 2.B is larger than the threshold value, CallDroid responds that two apps are suspected as cloned. A small threshold value implies that CallDroid tries to detect cloned apps aggressively. On the contrary, as the threshold becomes larger, CallDroid behaves conservatively, giving a positive answer only when there exist more similar parts.

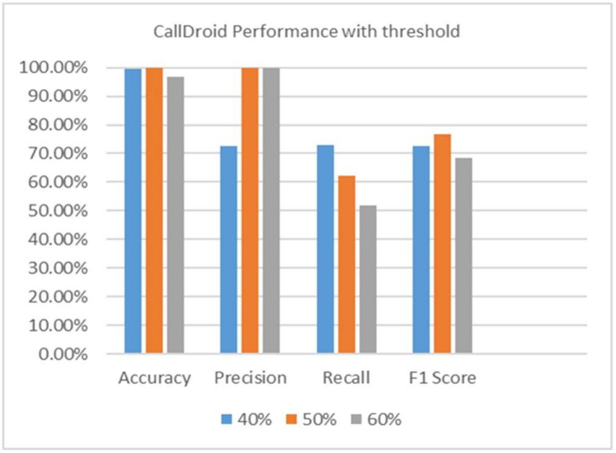


Fig. 4. CallDroid sensitivity on the similarity threshold

Fig. 4 reveals this tradeoff. In this experiment, we configure the similarity threshold values ranging from 40% to 60%, and measure four metrics, namely accuracy, precision, recall and F1 score. When we set the value as 40%, CallDroid shows the best recall value, a statistical metric that represents the ability to find the relevant cases within a dataset. However, this setting gives the worst precision value, another statistical metric that shows the ability to identify only the relevant apps. As a consequence, we can obtain the best F1 score, the harmonic mean of precision and recall, with the threshold value of 50%. Hereafter, experimental results reported in this paper is measured with this value (unless we mention another value explicitly).

B. Performance Comparison

We compare CallDroid with one of the state-of-the-art tools, called SimiDroid. SimiDroid is an open-source framework, designed for detecting repackaged/cloned apps. It is implemented with three types of plugins, method-level, component-level, and resource-level. In this comparison, we put to use the component-level plugin since this takes an approach comparable to ours.

Table 4 shows the experimental results. Both tools present high accuracy. In terms of the precision and recall, CallDroid gives better results, which eventually leads to higher F1 score. These enhancements are achieved by two features of CallDroid, making use of the implicit intent and smali code based investigation. In specific, SimiDroid relies on component names extracted from the AndroidManifest.xml, while CallDroid utilizes the method call information in a smali code at a component-level, which are more robust to obfuscation.

TABLE IV. Performance comparison between CallDroid and SimiDroid

	Accuracy	**Precision**	**Recall**	**F1 Score**
SimiDroid (Component-level)	99.73%	99.47%	60.64%	75.34
CallDroid	99.74%	100%	62.29%	76.76

TABLE V. PERFORMANCE ON THREE OBFUSCATION TOOLS.

	#Apps	Accuracy	Precision	Recall	F1 Score
ProGuard	184	99.83%	100%	75.24%	85.89
DexGuard	184	99.41%	100%	12.08%	21.56
DashO	252	99.93%	100%	89.83%	94.64
ALL	620	99.74%	100%	62.29%	76.76

C. *Performance under different obfuscation techniques*

Performance analysis results on three different obfuscation tools are given in Table 5. In this table, we observe that CallDroid shows reasonable performance for the apps obfuscated by ProGuard and DashO. However, it performs poorly for the apps obfuscated by DexGuard. Our analysis exposes that DexGuard applies the renaming obfuscation technique aggressively both for class name and method name including component class name, resulting in lower recall (in actuality, this phenomenon is also observed in SimiDroid). DashO also employs renaming obfuscation. However, it does not obfuscate the method name at a smali code which allows higher recall in our CallDroid.

V. RELATED WORK

SimiDroid[3] is a plugin-based framework for multi-level comparison of Android apps and can support the understanding of similarities/changes among app versions and among repackaged apps. Li et al. devised several similarity comparison methods as plugins for SimiDroid. Those methods have been borrowed from descriptions in the state-of-the-art literature, covering code-based similarity comparison at the statement level or at the component level, and resource-based similarity comparison. Since SimiDroid's objective is providing to the community an extensible framework for supporting the comprehension of similarities among Android apps, we have evaluated our technique compared with SimiDroid.

DNADroid[4] is a tool which detects Android app cloning based on the "semantic information" of disassembled code. It compares program dependency graphs (PDGs) between methods in candidate apps by using subgraph isomorphism algorithms. DNADroid could detect two variants of the same malware and have a very low false positive rate. It may overlook cloned apps due to false negative. Some program obfuscations can evade PDG-based clone detection, which is a fundamental limitation of DNADroid. Note that subgraph isomorphism algorithm is not scalable when one needs to deal with millions of apps [1].

Chen et al. [15] pointed out three characteristics that made app clones difficult to detect by existing techniques: a billion opcode problem caused by cross-market publishing, gap between code clones and app clones, and prevalent Type 2 and Type 3 clones. According to their study, existing techniques for detecting Android app clones achieved either accuracy or scalability, but not both. To achieve both goals, they used a geometry characteristic, called centroid, of dependency graphs to compute the similarity between methods (code fragments) in two apps. Then, they synthesized the method-level similarities and drew a Y/N conclusion on app (core functionality) cloning. However, they did not consider code obfuscation.

Crussell et al. presented a scalable approach and developed a tool called AnDarwin [16, 17] that could detect similar Android apps based on their semantic information. AnDarwin has four advantages: it avoids comparing apps pairwise, it analyses only the app code without other information such as the app's market, signature, or description; it can detect both full and partial app similarity; and it can detect library code and remove it from the similarity analysis. AnDarwin also detects similar code that is injected into many apps, which may indicate the spread of malware. Using AnDarwin, Crussell et al. found 88 new variants of malware and identified 169 malicious apps. AnDarwin is robust against all transformations that do not alter methods' PDGs, but less robust against obfuscations that alter methods' PDGs.

To detect visually similar Android apps, Sun et al. developed DroidEagle [1] which consists of two subsystems: RepoEagle and HostEagle. RepoEagle performs large scale detection on app repositories, and HostEagle is a lightweith mobile app which can help uses to quickly detect similar app upon download. DroidEagle is based on two *visual resources*: *layout files* and *drawable images* in an app package file. Layout file, defining user interfaces (UIs) of an app, contains a hierarchy of View and ViewGroup objects, and defines user interfaces of an app. Slight modifications of a layout structure can easily mess up the user interface, so hackers cannot easily obfuscate layout files.

ViewDroid [18] proposes feature view graph for detecting repackaged malware. The feature view graph is based on the calling relations between UIs. ViewDroid constructed view graph based on static analysis of the control flow relationship between the views within the app.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a novel app cloning/repackaging detection tool. For similarity measurement, it explores the method signature of an app like the existing detection tools. The unique feature of our proposal is that it considers only the component classes that receive implicit intents and extracts the method signature related to the classes at a smali code. We carry out a real implementation-based experiment using non-obfuscated apps and their obfuscated versions. Experimental results have demonstrated the effectiveness of our proposal.

There are two research directions for future work. The first one is extending CallDroid so that it can identify better for repackaged apps obfuscated by DexGuard. Currently we are considering other features that are robust to renaming obfuscation. For example, the component classes that receive explicit intents may be robust to renaming obfuscation. Therefore, we will utilize API calls in the classes that receive explicit intents externally as feature information. The second direction is detecting partial cloning based on isomorphism, which is essential for investigating third-part library abuse and fragmentary duplication.

ACKNOWLEDGMENT

This research was supported by (1) the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2019-2015-0-00363) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation), and (2) Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the MIST (no. 2018R1A2B2004830)

REFERENCES

[1] M. Sun, M. Li, and J. C.S. Lui, “DroidEagle: Seamless detection of visually similar Android apps,” in Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, 2015, pp. 9.

[2] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution” in IEEE Symposium on Security and Privacy (SP), 2012, pp. 95-109.

[3] L. Li, T. F. Bissyandé, and J. Klein, “SimiDroid: Identifying and explaining similarities in Android apps,” in IEEE Trustcom/BigDataSE/ICESS, 2017, pp. 136-143.

[4] J. Crussel, C. Gibler, and H. Chen, “Attack of the clones: Detecting cloned applications on Android markets,” in European Symposium on Research in Computer Security, Springer, Berlin, Heidelberg, 2012, pp. 37-54.

[5] Y. Wang, H. Wu, H. Zhang, and A. Rountev, “Orlis: Obfuscation-resilient library detection for Android” in IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems, 2018, pp. 13-23.

[6] M. Backes, S. Bugiel, and E. Derr, “Reliable third-party library detection in android and its security applications,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2016, pp. 356-367.

[7] Zhou, W., Zhou, Y., Jiang, X., & Ning, P, “Detecting repackaged smartphone applications in third-party android marketplaces,” in Proceedings of the second ACM conference on Data and Application Security and Privacy, ACM, 2012, pp. 317-326.

[8] H. Wang, Y. Guo, Z. Ma, and X. Chen, "WuKong: a scalable and accurate two-phase approach to Android app clone detection," in Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, 2015, pp. 71-82.

[9] ProGuard, https://www.guardsquare.com/en/products/proguard

[10] DexGuard, https://www.guardsquare.com/en/products/dexguard

[11] DashO, https://www.preemptive.com/products/dasho/overview

[12] f-droid, https://f-droid.org/

[13] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: detecting capability leaks of android applications." in Proceedings of the 9th ACM symposium on Information, computer and communications security. ACM, 2014, pp. 531-536.

[14] C. Zheng, et al. "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications." in Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2012, pp. 93-104.

[15] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on Android markets,” in Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 175-186.

[16] J. Crussell, C. Gibler, and H. Chen, “AnDarwin: Scalable detection of semantically similar Android applications,” European Symposium on Research in Computer Security, Springer, Berlin, Heidelberg, 2013, pp. 182-199.

[17] J. Crussell, C. Gibler, and H. Chen, “Andarwin: Scalable detection of Android application clones based on semantics,” in IEEE Transactions on Mobile Computing, vol. 14, no.10, pp. 2007-2019, October. 2015.

[18] F. Zhang, H Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection,” in the 2014 ACM conference on Security and privacy in wireless & mobile networks, 2014, pp. 25-36.