Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph

Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, Li Xie

Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph

Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie

State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, China
{sunxin508,Sophie.xuer,zxin.nju}@gmail.com,{maobing,xieli}@nju.edu.cn

Abstract. Recently smartphones and mobile devices have gained incredible popularity for their vibrant feature-rich applications (or apps). Because it is easy to repackage Android apps, software plagiarism has become a serious problem. In this paper, we present an accurate and robust system DroidSim to detect code reuse. DroidSim calculates similarity score only with component-based control flow graph (CB-CFG). CB-CFG is a graph of which nodes are Android APIs and edges represent control flow precedence order in each Android component. Our system can be applied to detect repackaged apps and malware variants. We evaluate DroidSim on 121 apps and 706 malware variants. The results show that our system has no false negative and a false positive of 0.83% for repackaged apps, and a detection ratio of 96.60% for malware variants. Besides, ADAM is used to obfuscate apps and the result reveals that ADAM has no influence on our system.

Keywords: Mobile Applications, Code Reuse, Repackaging, Malware Variants

1 Introduction

Smartphones have played a more and more important role in people's life due to abundant and feature-rich smartphone applications (or apps) that people can download and experience from app repositories such as Apple's App Store [2] and Google's Google Play [3]. Recent statistics show that till the second quarter of 2013, Android dominates the mobile device market with 79.3% of market shares while the next closest platform iOS accounts for 13.2% of overall share [1]. Now Google Play has officially reached over 1 million apps and it has finally outgrown App Store [4]. Since users are no longer satisfied with a few functionalities like making phone calls or sending messages, they are willing to browse and download apps which can meet their other various demands.

Code reuse occurs when different apps share the same code. It is often found in repackaged apps and malware variants.

Users browse and download apps from markets. Developers submit apps to markets to make them available to users and accordingly gain profits from submitted apps. Therefore, a healthy ecosystem comes into being. Unfortunately,

this ecosystem is mostly threatened by repackaged apps. A repackaged app emerges when a plagiarist unpacks a legitimate app, modifies certain code and redistributes it violating the intellectual property of original developer. Developers can directly charge for their apps, but many instead offer free apps and gain monetary profits from in-app billing or third-party ad libraries. Apps are repackaged for two motivations. First, a plagiarist can modify the ad library's client ID or embed new ad libraries to steal or re-route ad revenues [10]. Second, malicious payloads or exploits may be injected into popular apps to increase propagation. Once installed, this kind of apps can leak privacy, send messages to premium numbers and even turn the infected phones into bots. A recent work indicates that 86% of Android malwares repackage other legitimate (popular) apps [11], which is the main vehicle for propagation. Malware authors tend to enclose malicious payloads to as many apps as possible, which leads to different variants that should be classified into one malware family.

Due to the huge amount of Android apps, researchers have proposed several detection algorithms based on static analysis [12] [13] [14] [17] [18]. Till now, no dynamic algorithm has been proposed because dynamic detection is too slow to bear and Android specific input is hard to feed. While static analysis is fast, it is not robust enough to detect repackaged apps or malware variants especially when they are obfuscated. These algorithms generally sacrifice robustness for scalability. Simple obfuscation techniques can cause considerable false negative rates [15]. In practice, several obfuscation techniques have been used to successfully evade 10 state-of-the-art commercial mobile anti-malware products [16]. Without doubt, these obfuscation techniques can be easily applied to hinder the existing algorithms.

In this paper, we propose an accurate and robust system called DroidSim to effectively detect code reuse. DroidSim performs pare-wise comparison based on component-based control flow graph (CB-CFG). CB-CFGs are generated by static analysis for every component. Nodes in them are Android APIs and edges reflect the control flow precedence relationship of APIs. We propose CB-CFG based on three insights. First, Android APIs can represent semantic information. All the sensitive behaviors must interact with Android phone hardware through different APIs. Hence, Android APIs are effective to denote behaviors. Second, there are no superfluous APIs. It is hard to modify APIs without modifying the original behaviors. Third, the control flow can clearly clarify the relationship between two APIs. We notice that several problems occur when constructing CB-CFGs. Java inheritance and reflection usually make us ambiguous about which method is to invoke. Multi-threading and callbacks are also big challenges for static analysis.

We present DroidSim and apply it to detect repackaged apps and malware variants. For repackaged app detection, a dataset which contains 25 pairs of repackaged apps and other 61 irrelevant apps is customized from the Android Malware Genome Project [11] and the Internet. Our system shows that all the repackaged apps are detected and only 1 irrelevant app is falsely detected as repackaged, indicating a false negative of 0.00% and a false positive of 0.83%.

For malware variant detection, we evaluate our system on 706 variants and gain a detection ratio of 96.60%.

Our contributions are two folds:

- We propose a novel approach to detect code reuse in Android. DroidSim performs the pair-wise comparison and computes the similarity score only by CB-CFGs. CB-CFGs consist of Android APIs and reflect the relationship between different APIs. It can represent the semantic information of apps and is able to denote the behaviors of a component. Our system can be applied to detect repackaged apps and malware variants.
- Since an app has no superfluous APIs and the Android system has no redundant API to replace, we believe CB-CFG is not easy to modify and is able to resist common obfuscations. In our experiment, we leverage ADAM [22] to obfuscate the samples and the result shows that ADAM has no influence on our system.

The rest of the paper is organized as follows: section 2 introduces the system overview. In section 3, we describe the design and implementation for static code reuse detection, followed by evaluation in section 4. Section 5 presents the limitations of DroidSim and future work. Finally we describe related work in Section 6 and conclude in section 7.

2 System Overview

2.1 Threat Model

We aim to detect code reuse in Android apps. For repackaged apps, plagiarists usually don't modify the functionalities of legitimate apps, so the code must be very similar to the legitimate apps intuitively. A similarity score can be calculated to discern the repackaged apps. Malware variants in one family share the same malicious code snippets in most cases. And these shared code can be extracted as the signature of one malware family. With the evolution some obfuscation techniques are applied to repackaged apps and malware variants to escape detection. For instance, a real-world malware Gamex [6] has been found to apply some simple obfuscation techniques.

2.2 Assumption

In this paper we consider only *classes.dex* and the author information of an app. To detect repackaged apps, two assumptions must be satisfied. First, the signing key of a developer is not leaked. Once guaranteed, the repackaged app must be signed by a different key from the original one. Second, we measure the similarity score based on the DEX code and leave native code alone because native code only occupies a small portion of real-world apps and is much harder to modify than dalvik bytecode.

2.3 Methodology

DroidSim contains 4 steps as depicted in Figure 1. First, it pre-processes the whole dataset (section 3.1). For each app, it extracts two key features: *classes.dex* and the author information. *Classes.dex* is the main basis to calculate the similarity score while the author information is used to mark each app for excluding similarity comparison from the same author. Second, it constructs CB-CFGs from *classes.dex* (section 3.2). In general, Android apps consist of different components (e.g. Activity, Service, Broadcast Receiver and Content Provider). DroidSim generates CB-CFG for each component. Finally, it computes the similarity score according to the former features (section 3.3).

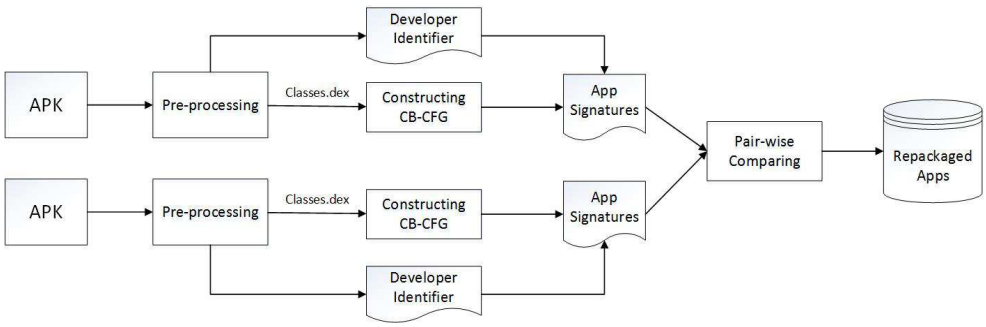

Fig. 1. Overview of our design

3 Design and Implementation

3.1 Pre-processing

Android apps are distributed in markets in the form of APK. An APK is a compressed archive of the program's Dalvik bytecode, resources and a XML manifest file. Two features are extracted first. One is *classes.dex* which contains the Dalvik bytecode for execution. The other is $META-INF$ which contains the detailed author information.

DroidSim utilizes *Keytool* utility [8] (included in the Android SDK) to extract the author information from *cert.rsa* in $META-INF$ and uses the public key as the identifier for each developer. In the following steps, this identifier will be integrated into signatures and used to determine whether two apps are from the same author.

For ease of constructing CB-CFGs, we leverage *baksmali* [7] to disassemble the DEX format. After disassembling, the DEX files are transformed to *smali* files. Disassembling is lossless. However, it may fail in rare circumstances. The *smali* files support the full functionality of the original dex format and are much

friendlier to read. DroidSim takes advantage of Android-apktool [5], a tool that already integrates *baksmali* and is able to decompress APKs. At the end of this step, DroidSim transforms an APK into a developer identifier and a set of *smali* files which will simplify the next few steps.

3.2 Constructing Component-Based Control Flow Graph (CB-CFG)

The second step is the core of DroidSim. It takes a developer identifier and *smali* files as input and outputs a signature for each app. Figure 2 shows the detailed process of how to construct a signature.

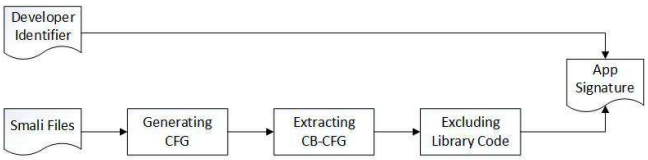

Fig. 2. The procedure of construction Component-Based Control Flow Graph

Generating Control Flow Graph DroidSim relies on the CFG that describes both intra-procedure control flow and inter-procedure control flow. Many tools are able to generate the intra-procedure CFGs. Unfortunately no existing tools can generate CFGs which consists of both intra-procedure and inter-procedure control flow. Therefore we implement a tool to generate this specific CFG based on the *smali* files.

To achieve this, DroidSim first identifies all the components in an app. For each method in a component, we divide the method body into many Basic Blocks. A Basic Block is a straight-line piece of code without any jump instructions or jump targets. It is easy to construct intra-procedure control flow. For inter-procedure control flow, method invocations in *smali* code always start with *invoke*∗ or *execute*∗ instructions. Resolving method invocations is hard when we meet the java characteristics such as polymorphism and inheritance. Besides, java reflection is also a challenge. It allows programs to invoke a method according to its string name. While it is easy to determine which method to invoke during runtime, it is not easy for static analysis. In this paper, we make a conservative approach that connects all the possible paths for reflections. Multi-threading and callbacks are often found in Android apps and hard to analyze for static analysis. In the same way, we conservatively connect all possible paths. Although conservative approaches may cause imprecision, we believe that it will make little influence because the same approach must be made for both the repackaged and the legitimate apps when meeting the same ambiguous method invocation.

Extracting CB-CFG CB-CFG is a graph of which nodes are Android APIs and edges represent control flow precedence relationship. It is extracted for each component. The node of CFG generated above represents a Basic Block and the directed edge represents control flow relationship. In this step DroidSim extracts APIs in Basic Blocks and omits other statements to generate nodes of CB-CFGs. If a Basic Block has more than one API, we divide it into different nodes in CB-CFG to make sure that each node has only one API. However, not all of Android APIs are suitable and only APIs that represent the app functionalities e.g. file operations, sending messages, making phone calls etc are reserved. We use APIs instead of Basic Blocks based on the insight that APIs usually have enough information to represent the program behaviors and are difficult to modify. The precedence order of different APIs is preserved in our CB-CFGs. Obtaining the precedence order is not easy because some Basic Blocks may have no APIs and more than one subsequent basic blocks. In this case, the precedence relationships should be stored until we find all the subsequent Basic Blocks which have APIs. Hence, a data structure is created to store the order and a depth-first traversal is implemented in the original CFG. Once nodes and edges are determined, CB-CFG is completed. Each node in CB-CFG corresponds to a unique type. And it is efficient to decide whether two nodes are of the same type when DroidSim calculates similarity score through subgraph isomorphism. Figure 3 is a simple example of CB-CFG generated from the *smali* code below.

Example of smali code

```
1. invoke-direct {v0, v1}, Landroid/content/Intent;-><init>(...;)V
2. if-gtz v17, :cond_0
3. invoke-virtual {v3, v4, v5}, Lcom/example/b/B;->display(II)I
4. goto/16 :goto_0
5. :cond_0
6. invoke-virtual {p0, v0}, Lcom/GoldDream/zj/zjService;->startActivity
   (Landroid/content/Intent;)V
7. :goto_0
8. invoke-virtual/range {v0 .. v5}, Landroid/telephony/SmsManager;
   ->sendTextMessage(...;)V
```

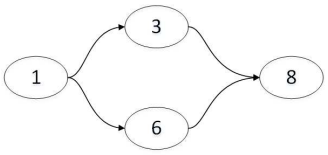

Fig. 3. Generated CB-CFG from smali code above. The content in each node is a digit for brevity. In real cases, it is the Android API.

Excluding library code Third-party libraries are often embedded in Android apps. For example, most free apps insert ad libraries like admob to compensate for their work. After pre-processing, these libraries are still reserved. In this case, computing the similarity score will likely bring about a misleading result due to the shared libraries. In case of it, DroidSim excludes the common library code by a white-list.

Ultimately DroidSim gets a set of CB-CFGs and a developer identifier for each app and integrates them into one signature for further detection.

3.3 Similar App Detection

For repackaged app detection, DroidSim performs pair-wise comparison for each app and calculates the similarity score by signatures. A signature contains a developer identifier and a set of CB-CFGs. The concrete process is as follows. For an app A, we want to detect whether app B is repackaged from A. DroidSim first checks whether two apps' identifiers are the same. If so, it neglects this pair, because they are written by the same author. Otherwise, it calculates the similarity score for B according to equation 1.

$$Sim(B) = \frac{\sum_{b \in B} |b|}{min(|A|, |B|)} \tag{1}$$

In Equation 1, the similarity score of B is calculated in comparison with A. In the denominator, the symbol $|A|$ represents the number of CB-CFGs in A and $|B|$ represents the number of CB-CFGs in B, $min(|A|, |B|)$ chooses the minor value in $|A|$ and $|B|$. In numerator, the symbol b stands for each CB-CFG in B. $|b|$ equals to 1 if b is subgraph isomorphic to any CB-CFG in A and equals to 0 otherwise.

For malware variant detection, an intuitive approach is proposed. The shared CB-CFGs are automatically extracted as signatures because variants of one family share the same malicious code. If an app contains the signature of certain malware family, it is regarded as the variant of this family.

Subgraph isomorphism is NP-complete. Fortunately because there are only limited nodes in CB-CFGs and each node has the unique API type, it is often efficient to perform subgrpah isomorphism mappings. And we leverage VF2 algorithm [19] to perform these mappings.

4 Evaluation

We have implemented a prototype system called DroidSim in Linux. In this section we first take a false positive and false negative measurement about detecting repackaged apps in a customized dataset. To elaborate the robustness, we measure how the prototype is resistant to several common obfuscation techniques. Then we leverage DroidSim to generate signatures for known malware families and apply these signatures to detect malware variants.

4.1 Experimental Setup

While it is easy to measure the false positive through manual verification, it is the opposite case for measuring the false negative. So we determine to customize a dataset from Android Malware Genome Project [11]. To measure the false negative, we randomly select several repackaged apps and find their corresponding legitimate apps via the Internet. What's more, we download irrelevant apps from a third-party market [1] to measure the false positive. For detecting malware variants, we also select 8 malware families from Android Malware Genome Project [11]. All apps are obfuscated by ADAM [22] to measure the obfuscation-resistance. The whole experiment is performed on a desktop PC with 3.4 GHz Intel Quad-cores CPU, 12GB RAM and Ubuntu 13.10 as OS.

4.2 Dataset Statistics

In total the dataset for repackaged app detection contains 121 apps. Among them, 25 are malwares from Android Malware Genome Project [11] and 25 are the corresponding legitimate apps. The rest 61 are downloaded from one third-party market. Callbacks are usually used in event handling process of widgets like buttons. Multi-threading is often applied in time-consuming tasks like network connecting and file downloading. Reflection is often used in malwares to escape detection. All the three specific features are included in our benchmark. In the first step, it takes 1214.87 seconds to generate all the signatures, in average, 10.04 seconds per app.

In practice, DroidSim removes CB-CFGs that are smaller than a specified size (< 5 nodes), because small graphs are more likely to be the same by chance. As illustrated on Figure 4, 95.9% of apps have less than 45 CB-CFGs. Among them, 57.8% have less than 15 CB-CFGs, which will reduce the frequence of comparison.

Figure 5 shows that 98.3% of CB-CFGs have more than 10 nodes. It is hard to get through subgraph isomorphism test by coincidence, which is also proved by our experiment that most of unrelated apps have 0 CB-CFG in common. Besides, 95.0% of nodes in CB-CFG are in the range between 10 and 70, which also guarantees the efficiency of subgraph isomorphism.

4.3 Repackaged App Detection

We measure the false positive and false negative on the dataset described in section 4.2. To measure the robustness of DroidSim, an obfuscation tool called ADAM [22] is used. ADAM is initially used to transform an original malware sample to different variants. It targets Dalvik bytecode and is often used as an automated obfuscation tool. The obfuscation techniques in ADAM include, (1) repacking such as realigning, re-signing and rebuilding APKs, (2) junk code insertion such as adding new methods that perform invalid operations, (3) method

[1] http://www.appsapk.com/

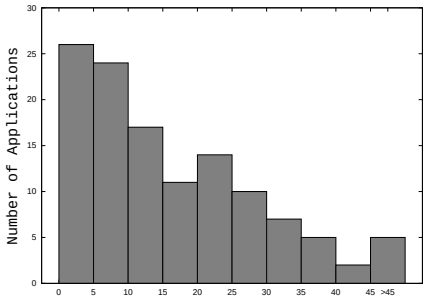

Fig. 4. Distribution of CB-CFG number

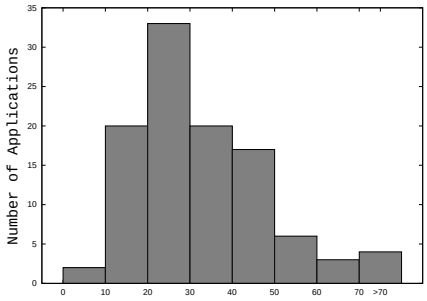

Fig. 5. Distribution of the average node number in CB-CFG

renaming, (4) code reordering such as inserting goto instruction to modify the control flow graph, (5) constant string encryption. There are also other obfuscation tools such as Proguard and Dexguard. However, Proguard directly targets java source code. Dexguard targets Dalvik bytecode, but is not free to get. Both tools are not adopted in our experiment.

Table 1. Experiment Result

Apps in Dataset	Detected before Obfuscation	Detected after Obfuscation
121	26	26

The first column in Table 1 indicates the number of apps in our dataset. The second column lists the number of apps detected as repackaged by DroidSim before obfuscation. The third indicates the number detected as repackaged after obfuscation. In our experiment, the threshold of similarity score is set to 0.3. If the similarity score is higher than 0.3, the app will be inferred as repackaged. We note that it is a tradeoff to determine the threshold. If we raise the threshold, the false positive will rise and the false negative will decrease.

Our manual verification shows that 25 apps are repackaged. Table 1 indicates that the false negative rate is 0.00% because all the repackaged apps have been detected successfully. And the false positive rate is 0.83%. 1 out of 121 is detected incorrectly. The mistaken one is named *xiangpeizhishu* [2]. After analyzing, we find that it only contains one Activity and its unique CB-CFG happens to be isomorphic to another app [3]. Most of the Android apps have more than one components and the APIs in them often vary a lot, so this case rarely happens in practice. As we mentioned in section 3.2, a conservative approach that connects all the possible paths is adopted when meeting java reflection, callbacks or multi-threading. And the experiment results show that this approximate approach can

[2] xiangpeizhishu. http://os-android.liqucn.com/rj/29261.shtml
[3] Free File Manager. http://www.appsapk.com/free-file-manager/

work well. For two similar apps, the constructed CB-CFGs remain the same because the same paths are connected when meeting the same ambiguous method invocation.

Table 1 also shows that ADAM has no influence on our experiment because DroidSim computes the similarity score only by CB-CFGs. Simple transformation techniques in ADAM such as repacking, methods renaming, and constant string encryption obviously do not change the control flow. As for junk code insertion, only inserting Android APIs can modify generated CB-CFGs, and thus lower the similarity score. Currently we don't distinguish the dead code in DroidSim and rely on a low threshold value to reduce the influence. It proves to be effective in our experiment for no existing obfuscation tool can do this. If a tool that automatically injects APIs as junk code is available, a semantic investigation must be employed to eliminate the unreachable code. With respect to code reordering, it can change the control flow. But the CB-CFG remains untouched because the precedence order of APIs is not modified. Overall, obfuscation techniques in ADAM have no effect on DroidSim.

4.4 Malware Variant Detection

Besides detecting repackaged apps, our system can also be used to detect malware variants. Previous work [11] shows that one malware family often has many variants. The variants share the same malicious code snippets, which can be the signature of this malware family. CB-CFG consists of Android APIs and their relationship, and is able to represent the functionality of shared code snippets. Hence we use DroidSim to extract a set of CB-CFGs as the signature of one malware family. To measure the validity and robustness, we analyze 8 malware families and extract signatures from several malware variants of each malware family. Then signatures are tested on 706 malware variants.

Table 2. Detection Results of Malware Variants

Malware Family	CB-CFGs (signature)	Samples	Detected before obfuscation	Detected after obfuscation	Percentage
AnserverBot	9	187	186	186	99.47%
DroidKungFu1	5	28	28	28	100.00%
DroidKungFu2	3	28	15	15	53.57%
DroidKungFu3	3	300	299	299	99.67%
DroidKungFu4	3	96	88	88	91.67%
DroidDream	2	14	13	13	92.86%
DroidDreamLight	1	41	41	41	100.00%
Zsone	1	12	12	12	100.00%
total	27	706	682	682	96.60%

As illustrated on Table 2, the first column lists the malware family name. The second column is the signature extracted by DroidSim. The third lists the

number of malware variants for test. The fourth indicates the malwares detected before ADAM is used. The fifth is the number of malware variants detected after ADAM is used. And the sixth represents the detection ratio. Among the 8 malware families, the detection ratio varies much. For AnserverBot, only 1 out of 187 escapes our detection. For further analysis, we notice that apktool [5] we employ does not generate the *smali* code corresponding to the signature. Thus it does not have the corresponding CB-CFGs. For the DroidKungFu series, only all variants of DroidKungFu1 are detected. The detection rate of DroidKungFu2 is the worst, which just exceeds a half. By analyzing the experiment log, we note that all the 13 missing variants are different versions of the same app named OnekeyVPN. Their behaviors are pretty different from the others. They do not access Wi-Fi state or open Internet connection while others do. Later we add the signature of this kind into the signatures, the detection ratio instantly rise to 100%. As for DroidKungFu3, one variant [4] escapes our detection. Manual analysis reveals that its malicious payload is modified exclusively for this app, which results in different CB-CFGs from others. 8 variants of DroidKungFu4 escape our detection. Among them, seven are different versions of the same app. All missing apps have different behaviors from the others. We believe that DroidSim can extract additional signatures to cover them if necessary. DroidDream shares the same situation with AnserverBot. Only one variant escape our detection due to the same reason with AnserverBot. As for the last two malware families, all their variants have been detected successfully.

5 Discussion

We try to detect code reuse by CB-CFGs. CB-CFG is essentially a graph. And we compare them with subgraph isomorphism. In DroidSim, we implement VF2 algorithm [19] to compare two graphs. Although subgraph isomorphism is NP-complete, it is still efficient because the node number is not that much and each node has a unique type. In our experiment, the average node number of 96.7% apps is less than 70. Hence the comparison is practical.

If an obfuscation tool that can automatically inject APIs as junk code is developed to attack our system, a semantic investigation on the program will be made to discern the unreachable code and drastically get rid of the influence.

DroidSim performs pair-wise comparison, indicating a time complexity of $O(n^2)$. Lots of algorithms try to decrease the time complexity and therefore sacrifice the robustness for scalability. In this paper, we focus on the robustness and the obfuscation techniques that may escape the detection of existing algorithms. To detect code reuse in a large scale, DroidSim should be implemented in a paralleled way and we leave it for future work.

[4] d99165d50a17d5678b13e0e7f70605f9fd4b7e9a.apk in Android Malware Genome Project [11]

6 Related Work

There are several approaches proposed to detect the repackaged apps. In general, these approaches can be divided into two categories: syntactic-level analysis and semantic-level analysis.

Juxtapp [20] is a scalable infrastructure for syntactic-level code similarity analysis among Android apps. It pre-processes the DEX files to obtain the opcode and discard the operands. Similarity score is calculated based on the feature vectors extracted every k-grams of opcode. And it is vulnerable to trivial obfuscation techniques e.g. injecting junk code every few instructions. DroidMoss [12] is also a syntactic-level detection algorithm. It generally employs the basic thoughts of MOSS [21], a well-known software similarity measurement algorithm. DroidMoss adopts fuzzy hashing to generate fingerprints from the opcode (excluding the oprands) for each app. It has the same shortcomings of Juxtapp that a consecutive opcode modification can easily escape the detection. [13] is a scalable algorithm that can detect the "piggybacked" apps. A "piggybacked" app refers to the app that plagiarists attach the malicious payloads in an independent package. It can't detect the payload in the original packages. What's more, it uses feature fingerprinting to represent the primary module, which is vulnerable to simple obfuscation techniques.

DNADroid [17] is more robust than the algorithms above. It uses the semantic information to detect repackaged apps. In detail, it uses PDG (only data dependency) to represent a method and computes pair-wise similarity score. Hence the robustness of DNADroid is equal to that of PDG. Obfuscation techniques such as adding data related variables can be utilized to change the data dependency in PDG, which will effectively cause false negative. AnDarwin [14] is the advanced version of DNADroid. It still uses PDG to uniquely represent a method. To meet scalability, it clusters similar apps based on the semantic vectors extracted from PDGs and thus is more vulnerable to obfuscation than DNADroid.

As various detection algorithms are proposed, [15] provides a framework for evaluating the obfuscation resilience. It applies this framework to evaluate AndroGuard [9] to find out that even simple obfuscation techniques can be applied to potentially cause false negative.

With respect to malware variant detection, a few static algorithms can work to some extent [13] [14] [20]. However, they all unintentionally find some variants when detecting repackaged apps. But our work can automatically extract the signatures and make a systematic examination on malware families. [23] proposes a malware variant detection algorithm in the desktop environment. It leverages the high-level models abstracted from the control flow graphs. This work differs from ours mainly in three aspects. First, our system can detect repackaged apps besides malware variants. Second, DroidSim works on a different platform Android. We develop a tool that constructs the control flow graphs from the *smali* code because of the lack of similar tools in Android. And some specific features like reflection, callbacks and multi-threading bring new challenges to

us. Third, Android malwares often implement the malicious payload in relatively independent components, using CB-CFGs is more efficient.

7 Conclusion

We present DroidSim, a tool that can detect code reuse in an accurate and robust way. In contrast with the earlier approaches, our system aims to detect code reuse in a more accurate way. DroidSim extracts CB-CFGs and author information to uniquely represent an app. It computes the similarity score based on CB-CFGs. We apply DroidSim to respectively detect repackaged apps and malware variants. Our results indicate that it can effectively detect the repackaged apps with no false negative and considerably low false positive rate. For malware variants, it automatically extracts signatures for 8 malware families and gains a detection ratio of 96.60% in average. All the results have demonstrated the effectiveness and robustness of our system.

Acknowledgments. We would like to thank the anonymous reviewers for their comments. This work was supported in part by grants from the Chinese National Natural Science Foundation (61073027, 60773171, 90818022, 61272078, and 61321491), and the Chinese National 863 High-Tech Program (2007AA01Z448, 2011AA01A202).

References

1. Android Nears 80% Market Share In Global Smartphone Shipments. http://techcrunch.com/2013/08/07/android-nears-80-market-share-in-global-smartphone-shipments-as-ios-and-blackberry-share-slides-per-idc/
2. Apple Inc. https://itunes.apple.com/us/genre/ios/id36?mt=8
3. Google Inc. https://play.google.com/store?hl=en&tab=w8
4. Android's Google Play beats App Store with over 1 million apps, now officially largest. http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest_id45680
5. Android-apktool. https://code.google.com/p/android-apktool/
6. Lookout Inc. Gamex Trojan. https://www.lookout.com/resources/top-threats/gamex-trojan
7. Smali - An assembler/disassembler for Android's dex format. http://code.google.com/p/smali/
8. Google Inc. Signing Your Applications. http://developer.android.com/tools/publishing/app-signing.html
9. AndroGuard: Reverse engineering, Malware and goodware of Android applications. http://code.google.com/p/androguard/
10. Clint, G., Ryan, S., Jonathan, C., Hao, C., Hui, Z., Heesook, C.: AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In: 11th International Conference on Mobile Systems, Applications and Services (Mobisys) pp. 431-444. ACM Press, Taipei (2013)

11. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: the 2012 IEEE Symposium on Security and Privacy (S&P), pp. 95-109. IEEE Press, Oakland (2012)
12. Wu, Z., Yajin, Z., Xuxian, J., Peng, N.: DroidMoss: Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In: 2nd ACM Conference on Data and Application Security and Privacy (CODASPY), pp. 317-326. ACM Press, San Antonio (2012)
13. Wu, Z., Yanjing, Z., Michael, G., Xuxian, J., Shihong, Z.: Fast, Scalable Detection of Piggybacked Mobile Applications. In: 3rd ACM conference on Data and application security and privacy (CODASPY), pp. 185-196. ACM Press, San Antonio (2013)
14. Jonathan, C., Clint, G., Hao, C.: AnDarwin: Scalable Semantics-Based Detection of Similar Android Applications. In: 18th European Symposium on Research in Computer Security (ESORICS), pp. 182-199. Springer Press, Egham (2013)
15. Heqing, H., Sencun, Z., Peng, L., Dinghao, Wu.: A framework for evaluating mobile app repackaging detection. In: 6th International Conference on Trust and Trustworthy Computing (TRUST), pp. 169-186. Springer Press, London (2013)
16. Vaibhav, R., Yan, C., Xuxian, J.: DroidChameleon: evaluating Android anti-malware against transformation attacks. In: 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS), pp. 329-334. ACM Press, Hangzhou (2013)
17. Jonathan, C., Clint, G., Hao, C.: Attack of the Clones: Detecting Cloned Applications on Android Markets. In: 17th European Symposium on Research in Computer Security (ESORICS), pp. 37-55. Springer Press, Pisa (2012)
18. Rahul, P., Andrew, N., Crisina, N., Xiangyu, Z.: Plagiarizing smartphone applications: attack strategies and defense techniques. In: 4th International Symposium on Engineering Secure Software and Systems (ESSoS), pp. 106-120. Springer Press, Eindhoven (2012)
19. Liugi, P., Pasquale, F., Carlo, S., Mario, V.: A (sub)graph isomorphism algorithm for matching large graphs. In: IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), pp. 1367-1372. IEEE Press (2004)
20. Steve, H., Ling, H., Edward, W., Saung, L., Charles, C., Dawn, S.: Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In: 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 62-81. Springer Press, Heraklion (2012)
21. Saul, S., Danial, S., Alex, A.: Winnowing: Local Algorithms for Document Fingerprinting. In: 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 76-85. ACM Press, New York (2003)
22. Min, Z., Patrick, P., John, C.: ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems. In: 9th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 82-101. Springer Press, Heraklion (2012)
23. Silvio, C., Yang, X.: Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs. In: 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 181-189. IEEE Press, Changsha (2011)