A Comparative Study of Android Repackaged Apps Detection Techniques

Xian Zhan[†], Tao Zhang[‡], Yutian Tang[†]
Department of Computing, The Hong Kong Polytechnic University[†]
College of Computer Science and Technology, Harbin Engineering University, China[‡]
chichoxian@gmail.com, cstzhang@hrbeu.edu.cn, csytang@comp.polyu.edu.hk

***Abstract*—Apps repackaging has become a serious problem which not only violates the copyrights of the original developers but also destroys the health of the Android ecosystem. A recent study shows that repackaged apps share a significant proportion of malware samples. Therefore, it is imperative to detect repackaged apps in various app markets. Although many detection technologies have been proposed, there lacks a systematic comparison among them. One reason is that many detection tools are not publicly available, and therefore little is known about their robustness and effectiveness. In this paper, we fill this gap by 1) analyzing these repackaging detection technologies; 2) implementing these detection techniques; 3) comparing them in terms of various metrics using real repackaged apps. The analysis and the experimental results reveal new insights, which shed light on the research of repackaged apps detection.**

I. Introduction

Recent years have witnessed a rapid increase in Android devices. According to a recent report from the statistia [1], Android has dominated the smartphone markets with a share of 85.0% in the second quarter of 2018. As of June 2018, Google Play has over 3.3 million apps available for downloading [2]. The prosperity of app economy has aroused great interest in cybercriminals and plagiarists to clone popular apps and republish them to obtain illegal profits. Cisco's annual security report [3] remarked that 99% of mobile malware targets on Android platform. Zhou et al. [4] showed that 86% of the malware samples were repackaged apps.

In light of this situation, many researchers proposed various approaches to detect repackaged apps. Since original apps and repackaged apps contain common code fragments, researchers first tried to adopt clone detection techniques to find repackaged apps. At the beginning, researchers used the hash values of opcode [5, 6] to identify the repackaged apps. However, this approach can be easily evaded by obfuscation techniques. To solve these issues, new approaches based on PDG (Program Dependence Graph) [7–9], CFG (Control Flow Graph) [10, 11] and AST (Abstract Syntax Tree) [12] were proposed. Since these methods leverage both syntax and semantic information of code, they are more resilient to code obfuscation than hash-based approaches. However, the above methods may be rendered ineffective because advanced code obfuscation techniques can change the control flow [13].

To mitigate the effect of code obfuscation, in addition to the code features, researchers tried to employ other features, such as the structural features of apps, resources features, and dynamic features to identify repackaged apps. In particular, new methods based on resources [14], UI features [15, 16], the variant CFG method (e.g., component-CFG) [17], or the centroid algorithm [10, 18] have been proposed to detect stealthy repackaged apps. These systems have different advantages and disadvantages, and we discuss the details in Section V.

To the best of our knowledge, there is not a systematic investigation and comparison of these detection methods yet. One reason is that many detection tools (e.g., [5, 6, 10, 11, 14, 15, 18]) are not publicly available, and therefore little is known about their robustness and effectiveness. To fill this gap, we first investigated existing detection techniques and proposed a taxonomy for them. Then, we implemented eight typical detection techniques [19] and defined a set of metrics for evaluating and comparing them. Finally, we prepared a dataset consisting of 15,297 app pairs of repackaged apps and the corresponding 2,776 original apps, then applied this dataset to evaluate these detection techniques. The analysis and the experimental results reveal new insights that are useful to the users of such detection methods and the researchers working on this direction. Our main contributions are as follows:

- To the best of our knowledge, we conduct the first systematic comparative study of existing repackaged apps detection techniques.
- We design a taxonomy for categorizing existing repackaged app detection techniques (Section IV) and analyze their advantages and disadvantages (Section V). We not only implement the detection techniques without publicly available sources, but also prepare a dataset consisting of real repackaged app pairs. Our implementations and the dataset are available here [19].
- We conduct extensive experiments to compare the existing detection techniques. The analysis and the experimental results reveal new insights that shed light on the research of this direction (Section V).

The rest of paper is organized as follows. Section II introduces the background knowledge of repackaged apps and detection techniques. Section IV describes the taxonomy of the repackaging detection techniques. Section V reports the evaluation results and findings. Section VI discusses the threats to validity and limitations. After introducing the related work in Section VII, we conclude our work in Section VIII.

II. Background

A. Repackaged Apps

Attackers usually utilize various reverse-engineering tools (e.g., apktool [20]) to disassemble an app and add some malicious payloads or third-party ads into the app or clone the code of the app. Then, they sign the reconstructed app using their own keys and publish this repackaged app to various app markets. Repackaged apps are usually related to plagiarism, and most repackaged apps may contain malicious behaviors, such as using premium services without user consent or stealing users' private information.

B. Repackaged Apps Detection

The process of repackaged apps detection generally contains five steps: 1) **Pre-Processing:** remove third-party libs and non-primary code (e.g., third-party libs). 2) **Feature Extraction:** transform the code to a suitable representation (e.g., CFG, PDG) and extract the features for comparison. We regard these features as the candidate fingerprints. **3) Filtering out Unnecessary Features:** To improve the calculation efficiency, we delete redundant features, e.g., operands [5, 6], or feature graphs whose nodes number is less than a threshold [7, 15, 17]. 4) **Similarity Detection:** compare the fingerprints by choosing the appropriate similarity comparison algorithm. 5) **Post-Processing:** analyze and verify the potential repackaged apps manually.

C. Repackaging Attack Models

According to the level of modification on the original APK files and the previous studies [15, 21], the repackaging attacks can be classified into three categories, including 1) **Easy Attack:** App pairs have the same source code. The slight differences only concern some simple changes like the package name, icons or the splash screen. 2) **Amateur Attack:** App pairs are different in some color schemes, language, images, and position. Some resources could be changed. Attackers change or add or delete a small number of functionalities. For example, malicious developers add some interactive views to the original apps. 3) **Grey Attack:** For this attack model, adversaries usually create a malicious app which simulates a popular app or inserts some malicious payloads into the original app. The attackers try to make use of the popularity of this app to spread the malware. This kind of apps only have a small part of the functions which are similar to a popular app, and it contains many additional sensitive behaviors. More knowledgeable Grey attacks just like piggybacked apps. The legitimate apps usually act as the carriers containing the malicious payloads [22, 23].

III. Literature Search

Based on the systematic literature review methodology proposed in [24–26], we search for related work published from 2007 to 2018 through IEEE Explore, ACM Digital Library, Springer Link and Google Scholar. We define the searching scope and set up the searching keywords. We also set up the exclusion criteria to delete the posters and papers that focus on other platforms. To avoid missing related work, we conduct a lightweight backward-snowballing mechanism on the remaining papers.

We collected 64 papers from top journals and conferences on app repackaging detection or similar apps detection, and exclude papers that mainly worked on how to protect apps from repackaged or described the characteristics of repacking attacks/repackaged apps or defense strategies or the impacts of repackaged apps. Eventually, we focus on 29 typical and essential tools in this paper. Most of the existing systems (77.5%) chose static analysis. There were only seven papers (17.5%) about dynamic analysis. The remaining papers (5%) adopted hybrid analysis.

IV. Taxonomy of Detection Techniques

In order to give readers a better understanding of the repackaging detection techniques, we classify these techniques based on source representation. The taxonomy method refers to the previous research [27, 28] on clone detection. Existing systems extract app features and use different techniques to convert app features into suitable source representations. Moreover, they adopt diverse algorithms for various source representations on different levels of granularity. Since the source representation affects the subsequent process, we group these detection techniques according to the source representation.

The classification is shown in the Table I. Each system is indicated by the tool name. If a system does not have a name, we use the first author's name to denote it.

TABLE I: The taxonomy based on the source representation

source representation	citation
opcode-based	DroidMOSS [6], Juxtapp [5], DroidKin [29], Shariar [30]
AST-based	Rahul [12]
token-based	WuKong [31]
PDG-based	DNADroid[7], Andarwin [8]
CFG-based	DroidSim [17], 3D-CFG [18]
API/method/class-level	DR-Droid [32], Kim [33], PiggyApp [9]
system-call	SCSDroid [34],PICARD [35],Lin [36]
traffic feature	Wu [37]
resources-based	FSquaDRA[38], ImageStruct [39]
UI-based	ViewDroid [15], DroidEagle [40], Soh [16], Malisa [41], RepDroid [42]
others	AndroidSOO [43],RepDetector [11],lili [22]
UI + code feature	MassVet [10]
resources + code feature	ResDroid [14]

A. Opcode-based Technique

DroidMOSS [6] uses a sliding window and rolling hash to cut the opcode sequence into different pieces and calculates the hash value for each piece of the opcode. The whole hash values are concatenated to represent an app. DroidMOSS employs edit distance for the similarity comparison to identify repackaged app pairs.

Juxtapp [5] uses the n-gram technique to get the same length of opcode and adopts the feature hash algorithm to generate the feature value as the fingerprint. Juxtapp employs the hierarchical clustering algorithm to find similar apps.

DroidKin [29] also adopts the n-gram to extract the code feature at the opcode and operands level. It utilizes the Simplifies Profile Intersection (SPI) [29] (a n-gram profile method) to calculate the similarilty score.

B. AST-based Technique

Rahul et al. [12] proposed an AST-based technique to detect repackaged apps. It extracts three features from AST, including 1) the number of arguments 2) direct and virtual invocation type and 3) syntactic artifacts of a method. It uses the k-nearest neighbor (KNN) algorithm for the similarity comparison.

C. Token-based Technique

Wukong [31] designs a two-phase (coarse and fine) approach to detect repackaged apps. The first stage is to achieve s-calability; Wukong applies the call frequencies of Android APIs as the fingerprints and employs the Manhattan distance for the similarity comparison. The coarse-grained phase can get some suspicious repackaged app pairs as the input in the fine-grained phase. The second stage is to achieve accuracy; Wukong exploits Boreas [44] (a token-based tool) to get the token feature of code fragments and uses the cosine similarity to find repackaged pairs.

D. PDG-based Technique

DNADroid [7] extracts the PDG as the fingerprint and uses the VF2 as the similarity comparison algorithm to identify repackaged apps. Andarwin [8] constructs the semantic feature from the PDG and uses the Locality Sensitive Hashing (LSH) to find repackaged apps.

E. CFG-based Technique

DroidSim [17] adopts the component-based control flow graph (CB-CFG) as the fingerprint and employs the VF2 as the similarity comparison algorithm to find repackaged apps.

3D-CFG [18] designs a new centroid algorithm, which utilizes the geometry characteristic of dependency graphs to measure the similarity between two apps. This system distills and converts CFG into a vector and constructs a database to store the features of apps. It employs the binary search to find repackaged app pairs.

F. API-based Technique

DR-Droid [32] identifies repackaged malware by recognizing the code with dependency relationship (class and method dependency). DR-Droid first distills the Class Dependence Graph (CDG) based on the call, data and Inter-component communication (ICC) dependencies. For the fine-grained processing, DR-Droid extracts the Method Call Graph (MCG) from the CDG. After that, DR-Droid distills the user-related functions and graph-related features, sensitive APIs and permissions from the MCGs as the fingerprint. Finally, it uses four binary classification algorithms (i.e., KNN, Decision Tree, Random Forest, and SVM) to detect repackaged malware.

PiggyApp [9] constructs PDG from the bytecode and employs the agglomerative clustering algorithm to decouple the primary code and third-party code. The authors deem that the large number of the same third-party libs can be clustered together. These clusters of third-party code are usually bigger than the others (the core functional code), and therefore they can exclude the third-party code. As for the primary code, PiggyAPP extracts the Android APIs and permissions as a feature vector. These feature vectors are organized into a metric space (Vantage Point Tree) and KNN is chosen to find the similar feature vectors.

Kim et al. [33] proposed a dynamic analysis method to detect plagiarized apps (a brunch of repackaged apps). They modified Android framework to custom the Android ROM. Apps to be tested will be executed in their custom sandbox. If an app invokes an Android API which is modified in the Android framework, the system can record the method call. It uses the Monkey [45] to trigger behaviors of an app and collects the invocation log. The log is a series of API calls, and they employ a sliding window to generate the fingerprint. Authors choose the Jaccard index for the similarity detection.

G. System Call based Technique

SCSDroid [34] is a dynamic analysis tool which extracts the thread-grained system call sequences of apps as the fingerprint to find the repackaged apps. The basic idea is that even though malicious repackaged apps can be camouflaged as benign apps, their malicious behaviours could be captured from the system call sequences. It uses the Bayes classifier to differentiate malicious apps and benign apps.

PICARD [35] also exploits system call analysis to capture and monitor the behavior of apps. It consists of two parts: a `common server` and a `client terminal`. The client is a PICARD app which is installed on mobile phones. The PICARD app can collect and monitor the execution traces of the monitored app and send messages to the `server` to check whether apps contain malicious behaviors.

The system [36] also adopts dynamic analysis. This tool mainly collects the framework-level event-related behaviors of apps. It works like a sandbox to get the system callback information and uses the Longest Common Sequence (LCS) to compare the tested apps with the malware signature to find the repackaged apps.

H. Traffic Feature based Technique

This system [37] employs the HTTP traffic as their fingerprint to detect repackaged apps. It uses the comparison algorithm based on the balanced Vantage Point Tree (VPT), which can remarkably improve the detection rate and meet the scalability requirements. The similarity comparison algorithm is the Hungarian algorithm.

I. Resource-based Technique

FSquaDRA [38] iterates the MANIFEST.MF file and gets the hash value of each resource file in an apk file. It calculates the Jaccard similarity score for tested apps and classifies them based on the resources similarity.

ImageStruct [39] is another resource-based repackaging detection tool. It gets the screenshots of an app and calculates the feature value by using the pHash algorithm. It uses the redis database to store image fingerprint and exploits the edit distance to find repackaged apps.

J. UI-based Technique

ViewDroid [15] extracts the View Feature Graph (Activity dependence relation) as the fingerprint and uses the VF2 to compare the similarity and identify repackaged apps.

DroidEagle [40] is a UI-based system which can conduct app repository analysis and host-based detection. Regarding app repository analysis, it extracts the layout tree as the fingerprint which can indicate a visual structure of an app. DroidEagle introduces Layout Edit Distance (LED for short) as a metric to measure the similarity between two layout trees. The second part called HostEagle conducts a real-time detection on mobile phones. HostEagle extracts a layout tree and calculates the hash value of the layout tree as the visual signature. By comparing the signature of the app in the remote server, it can decide whether this app is malware or not.

Soh et al. [16] proposed a dynamic detection method. This system uses the UIAutomator [46] to dump the view hierarchy of an app. The view information is stored as an XML file. Then the system parses the XML file, and each UI can be represented by 17 distinct attributes as a vector. They use the LSH to find the similar vectors, and employ the Hungarian algorithm to find the optimal activities pairs among these similar vectors.

RepDroid [42] also uses the UIAutomator [46] to dump the Layout trees of each app at the running time. Each layout feature can be represented as a feature vector. The similarity is quantified by the edit distance.

This tool [41] implements impersonation detection that mainly detect two types of impersonation: repackaging and phishing. The authors first design a UI crawler as a part of the Android emulator. According to the activity invocation relations and view hierarchy, it can get the screenshot of each Activity. Based on the screenshot similarity, the tool can find repackaged apps.

K. Hybrid Technique

ResDroid [14] uses a two-stage methodology which combines the resource features and UI features to detect repackaged apps. For the coarse-grained processing, it first generates a 15-dimensional from the resources of apps. ResDroid uses two clustering algorithms (nearest neighbor search and spectral clustering) to get a rough set of suspicious apps. For the fine-grained processing, ResDroid constructs the Activity Transition Graph (ATG) and employs the depth-first search algorithm to get the Activity sequence (AS). For each `Activity` from the AS, ResDroid parses the layout of Activity and gets the GUIs with corresponding Event Handlers' attributes. In this stage, the Layout Feature sequences and Event Handler sequences are considered as fingerprints. Finally, ResDroid uses the LCS to identify the repackaged apps.

MassVet [10] is a system to catch Android malware, especially for the malicious repackaged apps. The authors devised an original algorithm named centroid [18] to get the v-core and m-core of each app. A v-core is a geometric center of a view graph. A m-core is a geometric center of the CFG. The first-step analysis is used to identify apps sharing the similar view structure. When two apps own the similar view structure, a differential analysis will be conducted by comparing the m-core to find suspicious code segments (method level).

L. Other Techniques

AndroidSOO [43] can effectively find repackaged apps based on the `String Offset Order`. They find that the repackaging tools can easily access the .smali code, but it is difficult to change the string portion of the data section in alphabetical order. By comparing the constant string and offset of dex file, they can identify whether an app is repackaged or not.

Guan et al. [11] proposes a semantic-based approach named RepDetector which extracts input-output states of core functions with state flow graphs (SFGs) in apps and compare the function similarity to decide whether the app is repackaged or not. The authors introduce the Mahalanobis distance to compute the similarity.

Li et al. [22] propose an approach to decouple malicious payloads from piggybacked apps. It uses the package dependency graph as the main feature and sets up a series of metrics to distinguish malicious payload.

V. Analysis and Comparison

It is sometimes difficult to infer from a paper how well a tool works in practice. The algorithm might has unknowingly advantages or disadvantages to repackaged app detection systems, and you can find the case in subsection V-C. Researchers might have neglected the dataset with special features, and you can see an example in **Finding 8**. In other cases, some new characteristics of apps or Android platform might affect the effectiveness of existing tools. We will discuss these cases in the following subsections. To understand different repackaged apps detection techniques, we devise a series of experiments to evaluate them.

We conduct the experiments to answer a series of research questions (**RQs**). For each research question, we introduce the **Motivation** why we conduct the corresponding comparisons or analysis and the **Approach** used to conduct the experiment. We show the experimental data in **Result** and list the **Findings** from the experimental results. Moreover, we summarize the answers to the **RQs** in **Conclusion**.

A. Experimental Setup

To ensure consistency and fairness in the experiment, we use the same data set to evaluate repackaged apps detection systems. More precisely, we use the apps crawled from Androzoo [47, 48], which is an online app repository including more than 8 million apps. It also provides a benchmark for repackaged app analysis, which include 2,776 original apps and 15,297 repackaged apps. We use these apps as the ground truth because it contains both small-size and large-size apps and covers 40 distinct types of repackaged malware families.

Among these repackaged pairs, 100 repackaged pairs have been manually verified as the real repackaged pairs. All experiments were conducted on a PC running Ubuntu Linux 14.04 LTS with an 8-core Intel i7 3.70 GHz CPU and 32 GB memory. We implement and evaluate these typical systems one by one and report their performance. Due to page limit, we only report part of the results and the other results can be found from this link [19].

B. Common Feature Analysis

RQ1: What do the existing repackaging detection techniques have in common?

Motivation: Through comparing different detection techniques, we tried to find new insights to have a clearer understanding of them.

Approach: We read the collected papers and extracted the common parts of these research.

Finding 1: Existing repackaging detection systems have some common assumptions.

The details are as follows: 1) Remove the third-party libs. If third-party libs are reserved, the code from third-party libs may cause false negatives or false positives. For instance, irrelevant apps contain many identical libs, which may lead to false positives. If the original apps and repackaged apps contain many different libs, this may lead to false negatives [49]. Taking an example from our experiment, we implement the prototype of MassVet which use the whitelist to remove the third-party libs. The false positive rate of this system is 5.5%, and one of the reasons is that the whitelist cannot filter out all the third-party libs.

2) Repackaged apps and original apps have different signatures (certificates). If a developer updates his app by adding some new functions, we cannot say this app is repackaged. The repackaged apps must come from different developers with different certificates. All studies assume legitimate developers do not disclose their private keys (for apps signing) to others. In fact, some private keys were leaked or stolen [50]. This assumption may introduce some false negatives. This rule mainly excludes the updated apps.

3) Ignore the native code of an app. According to the original papers, the datasets used by most previous work only have tiny parts of apps containing the native code and most apps' essential functions are built on the Android Framework level.

Finding 2: Different detection systems employ different methods to remove third-party libs, and the most common approach is to use whitelist.

Table II lists how different systems filter out third-party libs. It can be seen that the majority of systems apply the whitelist to filter third-party libs. The proportion accounts for 42.5% of the total number. Tthere is a drawback in using the whitelist to filter out public libs because the whitelist cannot include all third-party libs. Besides, if the names of libraries are obfuscated by some tools like DexGuard [51], it is difficult to filter them out by searching the package name.

TABLE II: A BRIEF COMPARISON OF HOW DIFFERENT DETECTION APPROACHES FILTER OUT THE THIRD-PARTY LIBRARIES

Method	Tools
No mentioned	DR-Droid [52],HookRanker [53],Lin[36] RepDroid[42], RepDetector[11], [54], ImageStruct[39],SCSdroid[34],Shariar[30]
Whitelist	DroidMOSS [6],PICARD [35],Soh [16], MassVet [10],DroidEagle [40],Wu [37], ViewDroid[15],DNADroid[7], Juxtapp[5],DroidSim[17],3D-CFG [18] FSquaDRA [38], Andarwin [8]
callback functions	SimiDroid [55]
UIs/ publisher identities/ APIs	Kim [33]
clustering-based method	Wukong [31], PiggyApp [9]
page rank	ResDroid [14]

A few tools adopt the clustering algorithm to find the third-party libs, because the same libs have the same features so that they could be clustered together. Based on this idea, Wukong [31] and PiggyApp [9] use the clustering-based method to find the third-party libs. Kim et al. [33] point out that the ad libs have some special APIs features or unique UIs; they can remove ad-libs by using these features. ResDroid exploits the PageRank algorithm to delete the primary code [14].

C. Comparative Analysis

RQ2: What are the similarities and differences among the detection techniques that have some parts in common?

Motivation: We found that some systems adopt the same code representation but different similarity comparison algorithms while some techniques employ the same comparison algorithm but different code features. Therefore, we conducted a set of cross-system comparisons to find some new insights.

Approach: We divided the comparative experiments into two groups. One set of experiments are based on the same code representation, and the other set of experiments are based on the same comparison algorithm. We proposed different sub-research questions for the two set of experiments and conducted a series of comparative experiments on these detection tools.

RQ2-1: Some detection techniques use the same similarity detection algorithm but different source representation. What are the similarities and differences among them?

Motivation: Based on the previous description, it can be found that DNADroid, DroidSim, and ViewDroid employ subgraph isomorphism algorithm to identify repackaged apps. The difference among them is that these tools choose different code features. DNADroid selects PDGs as the fingerprint while DroidSim and ViewDroid choose CFGs and view feature graphs (VFG), respectively.

Finding 3: The number of nodes affects the detecting results and performance. If the number of nodes in a graph is too small (less than 3), this graph is possible isomorphic with many graphs. If the number of nodes in a graph is too large, it will take a long time to get the comparison results. Our experiment results show that if the number of nodes is more than 100, the VF2 algorithm will cost plenty of time to find subgraph isomorphic apps.

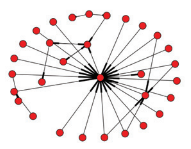
DNADroid defines filter rules to reduce unnecessary comparisons because a graph with a smaller number of nodes is more likely isomorphic to other graphs. It removes PDGs whose number of nodes is smaller than 5. The filtering rule excludes 25.43% of the total PDGs in an app. However, the filter rule deletes about half of the apps in our data set. DNADroid just can handle 8,773 repackaged app pairs in our data set. Without a doubt, the comparison algorithm limits the detection rate of DANDroid.

For DroidSim, it also neglects the CB-CFGs whose number of nodes is smaller than 5. Therefore, there are 35 app pairs that DroidSim cannot handle due to the small number of nodes. In our experiment, it costs about 159.86 hours to finish all operations. The average time to deal with each app is 37.73s. The shortest execution time is 0.16ms and the longest execution time is 18.94 hours. There are seven repackaged app pairs pairwise similar comparison time more than 10 hours. The longest similarity calculation time takes about 6 hours. Besides, there are six apps comparison time more than 2 hours but about 98.8% of apps comparison time is less than 100s. The apps with larger size usually lead to a longer comparison time. If CFGs contains more nodes, the VF2 will cost more time to decide whether the CFGs are isomorphic or not.

Finding 4: The structure of the graph can affect the performance of the similarity comparison algorithm. Figure 1(a) and (b) give an example of the VFG of the original app and repackaged app, respectively. Both of them are directed graphs. The VFG reflects the invocation relations of Activity. We use static analysis to get the VFGs and use NetworkX [56] to draw them. Each red dot represents an Activity, the edge between two nodes indicates the invocation relationship. The node pointed by the arrow indicates the called Activity. If many nodes point to one node, it will take a considerable amount of time to compare the similarity of two View graphs, even though the number of nodes in a graph is not very large.

Figure 1(a) and (b) is this situation, VF2 will cost longer time to attain the final result. In our experiment, it on average takes about 1.05ms to conduct a graph pair comparison. There are nine VFG pairs in our dataset having the same situation. The number of these apps accounts for 0.061% of the total number of test samples. The longest comparison time takes about 6 hours and the shortest time is 7.50s. The average comparison time is 3301.87s for these abnormal apps. This problem also exists in DNADroid and DroidSim since they all use the same algorithm VF2. For DroidSim, the longest comparison time is 19.84 hours.There are seven apps comparison time more than 10 hours.

Finding 5: The comparison algorithm cannot find the repackaged app pairs if their feature graphs are not isomorphic. If two graphs have the same nodes but different edge relationships, these two graphs may not be isomorphic. The VF2 algorithm may be invalidated if the repackaged apps change the invoking relationships of activities or add some new activities into the original apps.


(a) original apps

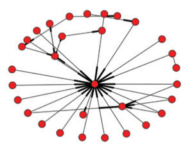

(b) repackaged apps

Fig. 1: An example of the View Feature Graph from a repackaged app pair.

For DNADroid, there are 19 repackaged app pairs getting a zero similar value by using the VF2 algorithm as the similarity detection algorithm. We manually checked these 19 apps and found these repackaged apps modified the data dependency relations of corresponding original apps, which cause their PDGs to be not isomorphic.

We manually checked the 91 app pairs which ViewDroid cannot identify them as the repackaged app pairs. We found that some invoking relationships among different activities are changed, or some repackaged apps add new activities which make them not isomorphic to the original one. Figure 3 gives an example of a repackaged app pair whose Activity relations have been changed.

Finding 6: Given the feature granularity, the VF2 algorithm has different effects (performance and detection rate) on these three detection systems. It has the most impact on ViewDroid and the effect on DNADroid and DroidSim is similar.

In most situations, an app can generate one VFG while an app usually contains massive functions which means a set of PDGs or CFGs can represent an app. The feature granularity of ViewDroid is much larger than that of DNADroid and DroidSim. If one app just has one VFG and the VFG of the repackaged app is not isomorphic to the VFG of the original one, ViewDroid will miss this repackaged app pair. Even though malicious developers insert some statements into an original method which leads to the change of a PDG or a CFG, this change is still small when they compare with ViewDroid. DNADroid and DroidSim use a set (PDGs and CFGs) to represent an app while ViewDroid just uses a VFG to represent an app. Table III shows the comparative results of detection rate, false negative rate (FNR), false positive rate (FPR) and average execution time of these three systems.

TABLE III: THE COMPARATIVE RESULTS OF DNADROID,DROIDSIM AND VIEWDROID

system	Detection Rate	FNR	FPR	Avg. execution time
DNADroid	97.7%	2.31%	0.0%	0.49s
DroidSim	99.8%	0.63%	0.13%	37.73s
ViewDroid	69.5%	0.85%	0.25%	38.89s

Conclusion

1) If VF2 is selected as the similarity comparison algorithm, the number of nodes in a graph and the shape of a graph can affect the performance of the whole system. 2) The VF2 algorithm cannot detect the repackaged app pair if the graph of the code representation of the original app is not isomorphic to the repackaged one.

Conclusion

3) In the case of the same comparison algorithm, the granularity of the fingerprint can affect the accuracy of the system. The larger the granularity, the lower the accuracy may be while the smaller the granularity, the higher the accuracy may be.

RQ2-2: Some detection techniques employ the same code representation but different detection techniques. What are the similarities and differences among them?

Motivation: Our summary shows that ViewDroid, ResDroid and MassVet all use the VFG as their features. ResDroid calls the Activity invocation graph as Activity Transition Graph (ATG) while ViewDroid and MassVet call the Activity invocation graph as VFG. ViewDroid employs the VFG as the only fingerprint to identify repackaged apps. Besides the ATG, ResDroid also adopts resources and event handlers as the code feature. MassVet combines UI feature (VFG) with CFG (code feature) to pick out repackaged apps. MassVet is a metric-based technique which use centroid algorithm [18] to represent the VFG (ATG) and CFG. Each graph will be transformed into a three-dimensional vector.

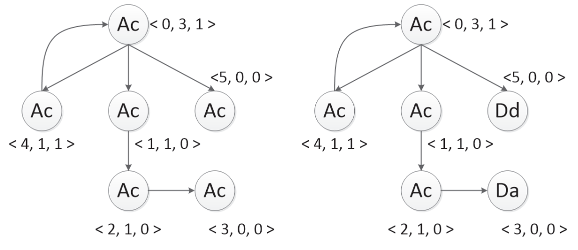
Approach: We applied ResDroid, ViewDroid and MassVet to the same dataset and conduct a series of comparative experiments to obtain the results.

Finding 7: Compared with the other systems, ViewDroid is most affected by the number of nodes and the structure of nodes. One reason is that the feature granularity of ViewDroid is coarser than the other two systems, and the other reason is that it just use a single feature (VFG) as the fingerprint while MassVet uses both UI features and CFGs and ResDroid uses both UI features and resources features.

For ViewDroid, if a graph with one node or two nodes could be isomorphic to many graphs, ViewDroid considers that the two apps are similar. In this case, it could cause false positives. ViewDroid first filters some ATGs whose number of nodes of a graph is smaller than two. Since a graph with a smaller number of nodes can match many graphs, it may increase the false positive rate. There are 4,657 app pairs whose nodes number is small than 3 in our dataset, and ViewDroid filters all of them. For our data set, the coverage rate of ViewDroid is about 69.6%. In contrast, ResDroid will not be affected by the number of activity node, and its coverage rate is 100%.

MassVet also has a limitation in finding similar view features, if the apps only have one or two Activity nodes. In our data set, there are 2,216 apps with the same centroid. Considering this problem of the small apps (an app with one or two Activity), MassVet proposes a solution that enriches the VFG by adding other UIs which can show up in a view. These UIs are considered as nodes for the graph.

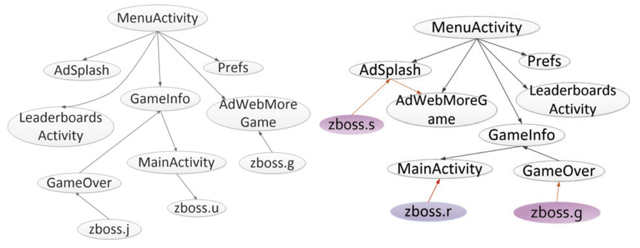
However, this will lead to two problems. 1) Some apps with few activities may not have some UI element like Dialog. 2) The solution might cause false positives. Figure 2 shows two different view graphs of two different apps. Figure 2(a) has five activities while Figure 2(b) has only three activities and two Dialogs. MassVet [10] defines a three-dimensional vector $\vec{c} =< \alpha, \beta, \gamma >$ to indicate a node in a view graph. Here, α is a sequence number assigned to each node in a view graph. The second element β is the out-degree of the node. The third element γ is the number of "transition loops" that the current node is involved. In this situation, MassVet considers them as the similar apps because they have the same centroid value. Considering these cases, MassVet can handler more apps than ViewDroid.



(a) view graph 1 (b) view graph 2

Fig. 2: Two View-graph examples
Ac:Activity; Da: AlertDialog; Dd: DatePickerDialog

The above analysis shows that the changes in the Activity invocation relationship have the most obvious impact on ViewDroid whereas they may not affect the detection result of ResDroid. We find that 91 app pairs are in this situation and ViewDroid cannot handle them. The false negative rate of ViewDroid is 0.85% . In contrast, ResDroid and MassVet can deal with this problem very well.



(a) original app (b) repackaged app

Fig. 3: An example of a repackaging pair whose ATGs are not isomorphic

Figure 3 shows a typical example of a repackaging app pair whose ATGs are not isomorphic while they are real repackaged app pair. The attacker changes some Activity relationship in this case. Table IV presents some basic information about the original app and repackaged app. According to Table IV, we can find the adversary changes the package name of the original app to "com.paopaoq.adm". Besides, a popular ad library "cn.domo" is inserted into the original app. Based on the Table IV and Figure 3, we can find that the repackaged pair has the same number of nodes, and most of the core components are the same.

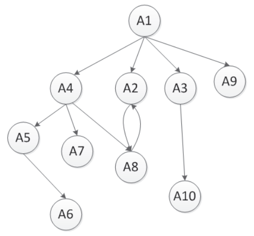
In this situation, we can find that the centroid value of the repackaged app is also changed. Whether MassVet can detect this types of repackaged app pair depends on the change degree and the threshold. ResDroid extracts ATG and exploits the depth-first search algorithm to get the Activity sequence. The change of ATG structure or the change of Activity invocation

TABLE IV: Some basic features of an example of the repackaging app pair

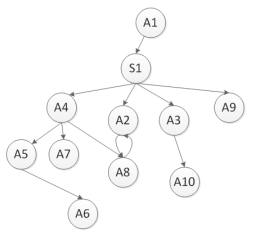
Type	original app	repackaged app
Package Name	com.mopower.game.zboss	com.paoapaoq.adm
Main Activity	.MenuActivity	com.mopower.game.zboss.MenuActivity
sha256 prefix	E2D75F1B	EA6A2D9E
Activities/Nodes/Edges	11 /11/10	13/11/11
third-party lib	no	cn.domob

relationship has less influence on ResDroid. In this example, ResDroid can find this repackaged app pair.

Finding 8: For some apps, they can have multiple separate ATGs. Only MassVet considers this situation and proposes a corresponding solution, and other two systems neglect this problem. These three systems all use the static analysis to get the ATG. In the Android system, Apps switch from one activity to another using `Intent` as a bridge. Since the activity transition can be obtained from analyzing the destination of `Intent`, data-flow analysis can be used to track the construction of ATG. However, the Activity can start a `Service` or a `Broadcast Receiver` by using `Intent`. The ATG generation process just considers the Activity invocation relationship. If an Activity invokes a Service or a Broadcast Receiver and then these components invoke other Activities, this situation can lead to separated ATGs.



(a) original ATG



(b) repackaged ATG

Fig. 4: An example of separated ATGs.

Figure 4(a) shows an original app's ATG. The node A1 stands for the main Activity, and each node denotes a `Activity`. The arrow stands for the transition direction. Figure 4(b) shows the repackaged app's components transition graph. An attacker inserts a service into the app, in this figure the node S1 stands for the new service component. The main activity starts the service, and the service invokes other activities. ResDroid just considers the transition between the activity so that we can get several ATGs. When the system traverses the ATG starting from the main Activity, ResDroid will get two separated graphs, and it just retains the graph which includes the main Activity. Therefore, ResDroid just compares the original graph with the separated graph with only one node, thus resulting in a false negative.

Conclusion

1) The number of nodes and the structure of the VFG have the most obvious impact on detection rate and performance of ViewDroid. 2) The change in the Activity invocation relationship has different degrees of impact on these three systems. 3) Only MassVet considers that an app sometimes can generate separated sub view feature graphs.

RQ3: Can resource-based technique effectively detect the repackaged apps?

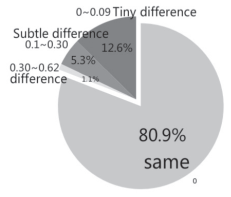
Motivation: We use FSquaDRA [38] and ResDroid to evaluate the effectiveness of resource-based technique. An app has various resources, and some resource can be easily changed. Compared with FSquaDRA, ResDroid not only considers the characteristic of resources but also adds the dependency relations of Activities, UIs on each Activity, and Event Handler associated with each UI and its corresponding properties as the fingerprints. We attempted to perform a detailed analysis of these features and tried to explore the impact of these features.

Approach: We extracted the resource features used by these two approaches and conducted a detailed comparison.

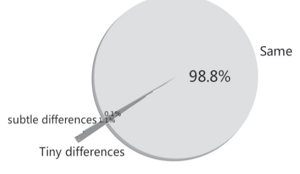
Result: According to the experiment, we find that adversaries sometimes can change the apps' color scheme, background pictures or icons. We extracted 15 resources that have the most significant impact on FSquaDRA from our dataset, and we found that 6,588 apps have different resource feature vectors from the original apps, which account for 43.1%(6588/15295) of the total number of repackaged apps. The detection rate of FSquaDRA is affected. When we set the threshold as 0.7, the detection rate just reaches 30.0%.

The accuracy of ResDroid is indeed higher than that of FSquaDRA. ResDroid can detect some types of amateur attack models which FSquaDRA cannot find. When we set the distance as 0.01, the detection rate of ResDroid can reach 98.1%. The false negative rate is 1.9%, and the false positive rate is 0.0%.

The main reason is that ResDroid chooses the resource is more difficult to change. Figure 5 shows the percentage distribution of Activity sequence distance and Event Handler distance. We can see from the Figure 5(a) and Figure 5(b) that the number of Activities and Event Handler seldom changed a lot compared with the resource in the FSquaDRA.



(a) Activity Sequence distance Distribution



(b) Event Handler Sequence Distance Distribution

Fig. 5: Pie graph of activity sequence and event handler sequence distance

Finding 9: FSquaDRA classifies some app pairs with many the same resources as similar while their code is not similar (i.e., false positives). The code of some app pairs are similar, but repackaged apps included many junk resources so that

FSquaDRA cannot identify these repackaged apps (i.e., false negatives).

Conclusion

1) FSquaDRA just uses resources as fingerprint without any code feature, which can be easily evaded by adding some junk resources into the apps. FSquaDRA cannot effectively detect amateur attack model if the resources changed a lot. However, it has good scalability. 2) ResDroid has higher performance than FSquaDRA.

RQ4: What are the differences among these detection systems?

Motivation: We have conducted cross comparisons for some systems. Besides, we also can compare them from other different perspectives. To give readers a clearer understanding, we conduct a high-level comparison on these systems.

Approach: We define a set of metrics for comparing different tools/techniques:

- ***Precision***: Precision is the ratio of correctly predicted positive samples to the total predicted samples. Precision is defined by $\frac{TP}{TP+FP}$, where TP = True Positive, FP = False Positive, FN = False Negative, TN = True Negative.
- ***Recall***: Recall is the ratio of correctly predicted positive samples to all samples in the actual class. Recall is defined by $\frac{TP}{TP+FN}$.
- ***Scalability***: To handle numerous apps from different markets, these tools should be capable of identifying the similar code from large code bank with high efficiency.
- ***Robustness***: The system should find different types of repackaging problems and can defend against some attacks (code modification and obfuscation)

In section II, we described three attack models. We can evaluate the robustness of these systems by determining how many attack models can be found. We also try to evaluate the scalability of different systems. In general, if a system uses pairwise comparison or dynamic analysis, the scalability of this system may be limited. At last, we give an intuitive comparison of whether these systems are resilient to code obfuscation. Generally, dynamic analysis can effectively defend against code obfuscation. We count the False Positive Rate (FPR) and False Negative Rate (FNR), when the value of the FPR or FNR is lower than the average, we consider this system has a good precision or recall.

Result: We first refer to the literature we collected and summarize an experimental comparison results from the literature based on different techniques.

Figure 6 shows some statistical information about our s-elected literature. The Figure 6(a) provides a dataset of the number of experimental apps from the detection tools. The biggest dataset comes form MassVet [10], and it tested about 1.5 million apps. Taking into account the issue of proportions, we did not draw all the systems in our chart. The smallest dataset comes from PICARD, and it just employed eight apps to test their system. PICARD is a dynamic analysis detection system. According to the size of the tested dataset, we also can roughly infer the scalability of some systems. The number of test samples for dynamic analysis is generally small.

Figure 6(b) provides an overall false positive rate (FPR) and false negative rate (FNR) of selected papers. The average value of FPR is about 3.8% and the average value of FNR is about 4.1%. The largest FNR(15%) and FPR(15%) comes from this system [33], following by DroidMOSS the FNR is 10.7% an opcode-based system.

The specific results are shown in Table V. The ✓indicates OK and $\times$ indicates NO. The symbol $\ominus$ that are used in attack modes indicates this system can handle some kinds of this repackaging attacks but not all of these attacks. D, S and H mean the dynamic analysis, static analysis and hybrid analysis, respectively.

Conclusion

1) The opcode-based method is not suitable for scalability and cannot deal with code modification or obfuscation. The public obfuscator can easily change the syntactic structure of code which can make these syntax-based methods fail. 2) Pairwise comparisons limit large-scale detection due to the combinatorial explosion of pairs to compare. Some systems use the subgraph isomorphism algorithm to identify repackaged apps, which is not good for scalability. 3) UI features and Resource features can be extracted by using both dynamic analysis and static analysis. Most UI-based systems adopt dynamic analysis, and the main features are Layout trees and screenshots. 4) Hybrid method (MassVet) combines the method level information and UI structure information, which can handle most repackaging problems.

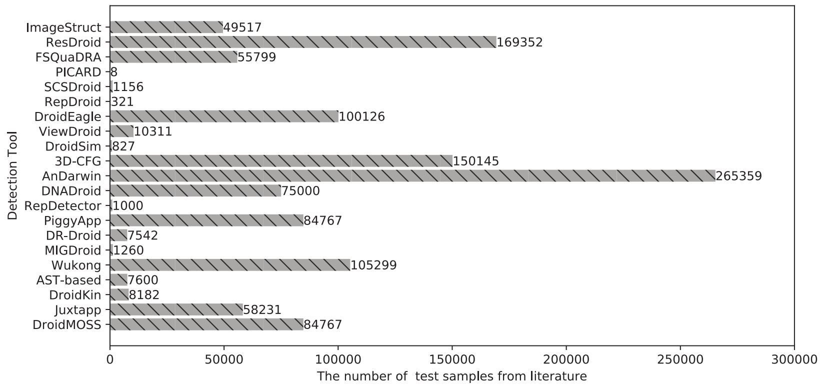
VI. Threats to Validity & Limitations

The literature set might not be complete. Another threat to validity comes from our experimental design. We just use the repackaged pairs in our dataset without other irrelevant apps. We do not verify whether the legitimate apps have any effect on the false positives. Besides, the threat lies in our dataset. This dataset just has 100 repackaged pairs that are manually verified. Based on our analysis, we found that some original apps in this dataset could be the repackaged versions.

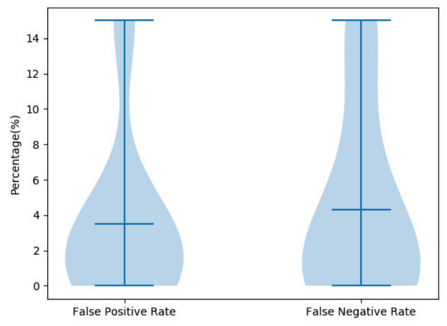
We give one taxonomy on repackaged app detection techniques in this paper while we also can classify these techniques from another perspective. We just analyzed some major repackaged apps detection systems and tools. Our future work will systematically compare more repackaging detection techniques and give a more comprehensive analysis.

VII. Related Work

The closely related area of repackaging detection is the clone detection [57] and the plagiarism detection [58]. Clone detection focuses on the intra-apps copies while the plagiarism detection focuses on the inter-application copies. Aakanshi et al. [59] summarized 16 techniques for code clone detection and analyzed 76 existing research articles. They investigated their approaches and provided different clone patterns. Roy et al. [27] investigated the state-of-the-art in clone detection and provided a review of the existing clone taxonomies and detection. The experimental evaluations of clone detection tools were also reported in this paper.



(a) The number of tested Android apps and its corresponding detection tool



(b) The violin plot of false positive rate and false negative rate from selected papers

Fig. 6: Some statistical data on our selected papers

TABLE V: A HIGH LEVEL COMPARISON OF DIFFERENT REPACKAGED APP DETECTION TOOLS

Detection System	Comparison Option						
	Easy Attack	Amateur Attack	Grey Attack	Analysis Method	Pairwise Comparison	Scalability	Code Obfuscation
DroidMOSS	✓	⊝	×	S	✓	×	×
Juxtapp	✓	⊝	×	S	×	⊝	×
DroidSim	✓	✓	⊝	S	✓	×	✓
3D-CFG	✓	✓	⊝	S	×	✓	✓
AST-based[12]	✓	✓	⊝	S	×	×	✓
Wukong	✓	✓	⊝	S	✓	✓	✓
DNADroid	✓	✓	⊝	S	✓	×	✓
Andarwin	✓	✓	⊝	S	×	✓	✓
DR-Droid	✓	✓	⊝	S	×	✓	✓
PiggyApp	✓	✓	⊝	S	×	✓	✓
[33]	✓	⊝	×	D	×	×	✓
SCSDroid	✓	⊝	✓	D	×	×	✓
PICARD	✓	⊝	✓	D	×	×	✓
FSquaDRA	✓	×	×	S	✓	×	✓
ImageStruct	✓	⊝	⊝	D	✓	×	✓
ViewDroid	✓	⊝	⊝	S	✓	×	✓
DroidEagle	✓	✓	⊝	H	✓	⊝	✓
[16]	✓	✓	×	D	×	×	✓
[41]	✓	⊝	×	D	×	×	✓
RepDroid	✓	⊝	×	D	✓	×	✓
ResDroid	✓	⊝	⊝	S	×	✓	✓
MassVet	✓	✓	✓	S	×	✓	✓

Bianchi et al. [60] investigated various GUI confusion attacks and proposed two novel approaches to defend against GUI confusion attacks. Luka et al. [61] presented a novel approach for detecting mobile app visual impersonation attacks. They extracted UIs from apps by using dynamic analysis and applied the screenshot to detect visual similarity.

Heqing et al. [62] analyzed three repackaging detection algorithms and devised a set of obfuscation algorithms to evaluate the obfuscation resilience of repackaging detection algorithms. Mojica et al. [63] investigated the problem of app reuse and summarized three categories of reuse. Gibler et al. [50] mainly focused on the features of repackaged apps and their impacts on the original developers.

All of these papers just briefly introduce typical repackaging detection techniques and basic knowledge, our work implements nine typical repackaged apps detection systems and identify their merits and demerits. There is no work for detailed classification of repackaged apps detection techniques. To the best of our knowledge, it is the first work to conduct a detailed classification and implement different types of repackaged app detection techniques. Besides, our work does not repeat all of the experiments, but we want to reveal how well these existing algorithms work in practice and what challenges were ignored by previous research.

VIII. Conclusion

We conduct a systematic investigation of existing repackaged app detection mechanisms and the corresponding literatures. We not only analyze their detection mechanism but also empirically evaluated them. According to the results, we report the advantages and disadvantages of each method. This study could faciliate users to select suitable mechanims for detecting repackaged apps and shed light on the research of detecting sophsistciated repacakged apps.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grant 61602258, in part by the China Postdoctoral Science Foundation under Grant 2017M621247, and in part by the Heilongjiang Postdoctoral Science Foundation under Grant LBH-Z17047.

REFERENCES

[1] "market share," https://www.statista.com/statistics/271496/.

[2] "Statista," https://www.statista.com/statistics/266210/.

[3] "Cisco," http://bgr.com/2014/01/21/android-mobile-malware-report/.

[4] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. S&P*, 2012.

[5] 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 *Proc. DIMVA*, 2012.

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

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

[8] J. Crussell, C. Gibler, and H.Chen, "Andarwin: Scalable detection of semantically similar android applications," in *Proc. ESORICS*, 2013.

[9] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. CODASPY*, 2013.

[10] 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 *Proc. USENIX Security*, 2015.

[11] Q. Guan, H. Huang, W. Luo, and S. Zhu, "Semantics-based repackaging detection for mobile apps," in *Proc. ESSoS*, 2016.

[12] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: Attack strategies and defense techniques," in *Proc. ESSoS*, 2012.

[13] C. K. Behera and D. LalithaBhaskari, "Different obfuscation techniques for code protection," *Procedia Computer Science vol.70*, 2015.

[14] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proc. ACSAC*, 2014.

[15] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM WiSec*, 2014.

[16] C. Soh, H. Tan, Y. Arnatovich, and L. Wang, "Detecting clones in android applications through analyzing user interfaces," in *Proc. ICPC*, 2015.

[17] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in android applications using component-based control flow graph," in *Proc. IFIP SEC*, 2014.

[18] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. ICSE*, 2014.

[19] https://github.com/MIchicho/Repackaged-APPs-detection.

[20] "android-apktool," https://code.google.com/p/android-apktool/.

[21] A. Kohli, "Decisiondroid: A supervised learning-based system to identify cloned android applications," in *Proc. FSE*, 2017.

[22] L. Li, D. Li, T. Bissyandé, D. Lo, J. Klein, and Y. Traon, "Ungrafting malicious code from piggybacked android apps," Technical report, SnT, Tech. Rep., 2016.

[23] L. Li, D. Li, T. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. Traon, "Automatically locating malicious packages in piggybacked android apps," in *Proc. MOBILESoft*, 2017.

[24] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. EASE*, 2014.

[25] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[26] "Rebooting research on detecting repackaged android apps: Literature review and benchmark," https://arxiv.org/pdf/1811.08520.pdf.

[27] C. Roy and J. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, Jan. 2007.

[28] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, May 2009.

[29] H. Gonzalez, N. Stakhanova, and A. Ghorbani, "Droidkin: Lightweight detection of android apps similarity," in *Proc. SecureComm*, 2014.

[30] H. Shahriar and V. Clincy, "Detection of repackaged android malware," in *Proc. ICITST*, 2014.

[31] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: A scalable and accurate two-phase approach to android app clone detection," in *Proc. ISSTA*, 2015.

[32] T. Ke, D. Yao, B. G. Ryder, and G. Tan, "Analysis of code heterogeneity for high-precision classification of repackaged malware," in *Proc. SPW*, 2016.

[33] D. Kim, A. Gokhale, V. Ganapathy, and A. Srivastava, "Detecting plagiarized mobile apps using api birthmarks," *Automated Software Engg. Vol.23*, Dec. 2016.

[34] D. Lin, Y. Lai, C. Chen, and C.Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Comput. Secur.*, Nov.2013.

[35] A. Alessandro, M. Fabio, S. Andrea, and S. Daniele, "Detection of repackaged mobile applications through a collaborative approach," *Concurrency and Computation: Practice and Experience*, Aug. 2015.

[36] Z. Lin, R.Wang, X. Jia, S. Zhang, and C. Wu, "Analyzing android repackaged malware by decoupling their event behaviors," in *Proc. IWSEC*, 2016.

[37] X. Wu, D. Zhang, X. Su, and W. Li, "Detect repackaged android application based on http traffic similarity," *Sec. and Commun. Netw.*, Feb. 2015.

[38] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, L. Spina, and E. Moser, "Fsquadra: fast detection of repackaged applications," in *Proc. IFIP DBSec*, 2014.

[39] S. Jiao, Y. Cheng, L. Ying, P. Su, and D. Feng, "Erratum: A rapid and scalable method for android application repackaging detection," in *Proc. IPSEC*, 2015.

[40] M. Sun, M. Li, and J. Lui, "Droideagle: Seamless detection of visually similar android apps," in *Proc. Wisec*, 2015.

[41] L. Malisa, K. Kostiainen, O. Michael, and S. Capkun, "Mobile application impersonation detection using dynamic user interface extraction," in *Proc. ESORICS*, 2016.

[42] S. Yue, W. Feng, J. Ma, Y. Jiang, X. Tao, C. Xu, and J. Lu, "Repdroid: An automated tool for android application repackaging detection," in *Proc. ICPC*, 2017.

[43] G. Hugo, K. Andi, S. Natalia, J. Alzahrani, and A. Ghorbani, "Exploring reverse engineering symptoms in android apps," in *Proc. EuroSec*, 2015.

[44] Y. Yuan and Y. Guo, "Boreas: an accurate and scalable token-based approach to code clone detection." in *Proc. ASE*, 2012.

[45] "Monkey," https://developer.android.com/studio/test/monkey.html.

[46] https://developer.android.com/training/testing/ui-automator.

[47] "Androzoo," https://androzoo.uni.lu/.

[48] "Repack," https://github.com/serval-snt-uni-lu/RePack.

[49] L. Li, T. Bissyandé, J. Klein, and Y. Traon, "An investigation into the use of common libraries in android apps," in *Proc. SANER*, 2016.

[50] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "Adrob: examining the landscape and impact of android application plagiarism," in *Proc. MobiSys*, 2013.

[51] "Dexguard," https://www.guardsquare.com/dexguard.

[52] K. Tian, D. Yao, B. Ryder, G. Tan, and G. Peng, "Detection of repackaged android malware with code-heterogeneity features," *TDSC*, Aug. 2017.

[53] L. Li, D. Li, T. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. Traon, "On locating malicious code in piggybacked android apps," *Journal of Computer Science & Technology*, Nov. 2017.

[54] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: attack strategies and defense techniques," in *Proc. ESSoS*, 2012.

[55] L. Li, T. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *Proc. ICESS*, 2017.

[56] "Networkx," https://networkx.github.io/.

[57] D. Baxter, A. Yahin, L. Moura, S. Marcelo, and B. Lorraine, "Clone detection using abstract syntax trees," in *ICSM*, 1998.

[58] B. Belkhouche, N. Anastasia, and H. Johnette, "Plagiarism detection in software designs," in *Proc. ACMSE*, 2004.

[59] A. Gupta and B. Suri, "A survey on code clone, its behavior and applications," in *Networking Communication and Data Knowledge Engineering*, Nov. 2017.

[60] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the android user interface," in *Proc. S&P*, 2014.

[61] L. Malisa, K. Kostiainen, M. Och, and S. Capkun, "Mobile application impersonation detection using dynamic user interface extraction," in *Proc. ESORICS*, 2016.

[62] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Proc. TRUST*, 2013.

[63] J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE software vol.31*, no. 2, Nov. 2014.