WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection

Haoyu Wang, Yao Guo, Ziang Ma, Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
{wanghy11, yaoguo, maziang14, cherry}@sei.pku.edu.cn

ABSTRACT

Repackaged Android applications (app clones) have been found in many third-party markets, which not only compromise the copyright of original authors, but also pose threats to security and privacy of mobile users. Both fine-grained and coarse-grained approaches have been proposed to detect app clones. However, fine-grained techniques employing complicated clone detection algorithms are difficult to scale to hundreds of thousands of apps, while coarse-grained techniques based on simple features are scalable but less accurate. This paper proposes WuKong, a two-phase detection approach that includes a coarse-grained detection phase to identify suspicious apps by comparing light-weight static semantic features, and a fine-grained phase to compare more detailed features for only those apps found in the first phase. To further improve the detection speed and accuracy, we also introduce an automated clustering-based preprocessing step to filter third-party libraries before conducting app clone detection. Experiments on more than 100,000 Android apps collected from five Android markets demonstrate the effectiveness and scalability of our approach.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering and reengineering*

General Terms

Algorithms, Security, Experimentation

Keywords

Clone detection, mobile applications, Android, repackaging, third-party library

1. INTRODUCTION

In the past few years, mobile devices such as smartphones and tablets have grown explosively. Android has since dominated the smartphone market, with more than 1.5 million devices activated daily [1]. A wide variety of feature-rich *mobile applications* (*apps* for short) have been developed. Currently, the number of Android apps in the Google Play market has surpassed the 1.4 million mark.

However, Android apps are easy to crack as many decompiling tools are available [5, 7]. Hackers can easily crack legitimate apps and re-advertise them in various third-party markets. Paid apps can be cracked and advertised for free. Hackers can also modify the ad libraries to steal revenues [26]. Malicious hackers may insert malware into legitimate apps to infect unsuspicious users [65]. These actions not only cause the original authors lose potential revenues, but may also compromise the security and privacy of mobile users.

Various techniques have been proposed to detect repackaged Android app clones, including techniques based on simple hashing [64, 27] or other static semantic features [63, 62], and also more complicated techniques based on PDGs [24, 25] or other existing code clone techniques [55].

We observe two key challenges in existing approaches:

- *How to achieve accuracy and scalability at the same time in detecting mobile app clones.* Although the simpler hashing-based approaches [64, 27] and static semantic feature based approaches [63, 62] are very fast, they are typically not as accurate as more complicated approaches such as PDG-based detection [24, 55]. On the contrary, although PDG-based approaches are more accurate, they cannot be scaled to perform app clone detection on a market with over a million apps. As a result, it is still a big challenge to develop an accurate and scalable approach to detect app clones on Android markets [21].
- *How to deal with third-party libraries during app clone detection.* Compared to desktop/server applications, one special characteristic of Android apps is that they typically contain a number of third-party libraries. Based on our evaluation, more than 60% of the sub-packages in Android apps are from third-party libraries, which might occur in many other apps in the exact same form. Detecting and filtering these third-party libraries is important, because if apps are dominated by these library code, app clone detection results will be skewed significantly.

  Most of the existing approaches [64, 27, 21, 24, 55] use a *whitelist* to filter external libraries by comparing their package names. For example, one most recent work [21] uses a list of 73 libraries in their whitelist. However, it is impossible to build a complete whitelist

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA

http://dx.doi.org/10.1145/2771783.2771795

of existing third-party libraries, as we found more than 600 different libraries with an automated clustering technique. Besides, some libraries have no specific package names or use obfuscated package names, which cannot be detected with a whitelist.

In this paper, we propose WuKong[1], a new approach to detect app clones on Android markets. We introduce two key techniques to achieve both accuracy and scalability.

- We propose *a novel clustering-based technique to identify and filter third-party libraries automatically*, which is more effective and accurate compared to the whitelist approach used in most state-of-the-art approaches. Experiment results show that this approach is more effective and removes more than 60% of all the sub-packages [2] in 100,000 Android apps studied.
- We propose *a two-phase detection technique to identify app clones,* which includes a *coarse-grained detection phase* to select suspicious cloned apps quickly by comparing their static semantic features, and a *fine-grained detection phase* to perform more detailed code segment level comparison to refine the results. Because the coarse-grained phase can quickly narrow down the suspicious clone pairs by several orders of magnitude, it allows the fine-grained phase to perform detailed detection on a limited number of clone candidates.

We have implemented a prototype system and studied more than 100,000 apps collected from five Android markets. We are able to perform app clone detection on these apps with no false positives, while the whole comparison process only takes several hours. Besides app clones from different markets, we also find that there exists a significant number of cloned app pairs within the same market as well.

2. BACKGROUND

2.1 Android Apps

Android apps are normally written in Java and compiled to Dalvik bytecode (DEX). All Java code is contained in the *classes.dex* file. The compiled code and resources are packaged as Android packages (.apk).

One particular feature of Android apps is that most of them contain third-party libraries, such as advertisement libraries, social network libraries, mobile analytic tools, etc. These libraries usually compose a considerable fraction of code (up to 50% in many apps), which may influence the accuracy and efficiency of app clone detection [63, 64].

Furthermore, many Android apps are obfuscated to increase the difficulty of reverse engineering [29, 61, 41]. For example, ProGuard [14] is an obfuscation tool that can be used to shrink, optimize and obfuscate the source code through removing unused code and renaming classes, fields, and methods with semantically obscure names. Obfuscation increases the difficulty of identifying third-party libraries.

2.2 App Clones

The term *app clones*, also known as *app repackaging* [64] or *app piggy-backing* [63], are used to describe the scenario when two apps have *similar core functionalities but different ownerships*. The core functionalities refer to the functional code, excluding the third-party libraries and frameworks. The ownership of an app is determined by the signature of the app, which is signed using the developers' private keys. We assume that the developers' private keys are not leaked, thus a cloned app must have a different signature. Apps developed by the same developer (different versions of the same app, for example) are not viewed as cloned apps.

Based on the features of Android apps and the criteria of app clones, we face the following challenges in detecting cloned apps:

- **How to identify the core functionalities of an app?** Only when the core functionalities of one app are cloned in another app, we can then consider them as a clone pair. Most apps contain third-party libraries, which may impact the detection result greatly. Identifying and filtering these external frameworks and libraries efficiently and accurately is the key to identify core functionalities.
- **How to perform pair-wise comparison efficiently?** App clones could appear on different markets or within the same market. Given apps from various markets, we should compare these apps pair-wisely to identify all the clone pairs. The number of comparisons grows explosively with the number of apps. For example, pair-wise comparisons of 100,000 apps will incur almost 5 billion comparisons (C^2_{100000}).
- **How to generate app features efficiently?** An app usually contains thousands of lines of code, with each app containing 50,000 op codes on average [27]. Generating precise features to represent each app is both important and difficult. Simplistic features may not be able to describe the detailed characteristics of apps, while detailed features may increase the complexity of comparison.

3. OVERVIEW OF WUKONG

3.1 Key Ideas

As stated above, we aim to achieve two important goals: one is to identify and filter third-party libraries accurately, the other is to achieve accuracy and scalability at the same time. To achieve these two goals, we propose two key techniques: clustering-based third-party library filtering and two-phase app clone detection:

(1) An automated clustering-based approach to filter third-party libraries efficiently and accurately. In general, third-party libraries have two characteristics: (1) *they are used by app developers without modification*, thus the same library in different apps possesses identical features; (2) *they may be used by many apps*, thus we can identify them via clustering with no prior knowledge. Based on these two characteristics, if we extract features at sub-package level for a large number of apps and cluster these features into groups, the sub-packages containing third-party libraries would be clustered into big groups because the same libraries are used by many different apps.

(2) A two-phase app clone detection approach to achieve accuracy and scalability simultaneously. Simple detection approaches could be scalable because they contain fewer features, but they cannot guarantee the same accuracy as more complicated approaches. While

[1]WuKong is the name of Monkey King, a famous fictional character in Chinese folklore, who can distinguish faked/cloned creatures and real ones using his naked eyes.

[2]In this paper, each sub-package refers to all bytecode files under a directory, excluding its sub-directories, such that there are no duplication in different sub-packages.

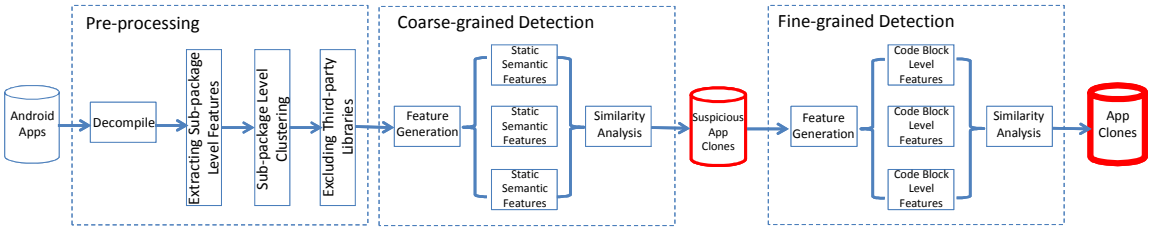

Figure 1: The overall architecture of WuKong.

complex approaches are more accurate because they can compare more detailed features, their performance becomes the bottleneck. With a two-phase detection framework, we can use the simpler approach to select clone pair candidates from millions of apps first, then use the complex approach to conduct detailed comparisons on a narrowed set of apps. In this way, we can expect to achieve accuracy and scalability simultaneously.

3.2 Overall Architecture of WuKong

The overall architecture of WuKong is shown in Fig. 1, which is comprised of three main stages.

In the *app pre-processing* stage, we preprocess each app to extract the intermediate SMALI [15] code and the developer keys. Meanwhile, we also filter third-party libraries using the proposed clustering-based technique, which is the key step to retrieve the core functionalities of each app. In the *coarse-grained detection* stage, we calculate the static semantic features for each app, identifying pairs of potentially cloned apps by comparing these features. In the *fine-grained detection* stage, code block level features are generated and similarity scores are calculated for each selected app pair (from the coarse-grained stage). In the end, app clone pairs are determined based on their similarity scores.

4. THIRD-PARTY LIBRARY FILTERING

As stated above, many apps include third-party libraries, which not only impact the accuracy of clone detection, but also slow down the detecting speed.

Third-party libraries may impact the accuracy of app clone detection in two ways. First, when two apps are not an app clone pair, but they use the same external frameworks or libraries, it could increase their similarity to a much higher level. Thus, it might lead to false positive result that they are incorrectly detected as clone pairs. On the other hand, considering that two apps are a clone pair and they have the same core functionalities, but the hacker may replace the original Ad libraries with other libraries, which could cause their similarity ratio to decrease. Thus, it might lead to a false negative result that the cloned apps evade detection.

4.1 Challenges

Most previous work uses a *whitelist* to filter third-party libraries by comparing the package names. However, our experiments show that whitelist-based filtering cannot filter external libraries effectively and accurately.

First, it is impossible to build a complete whitelist of existing third-party libraries. There are many other external libraries and frameworks besides the ad libraries used in most approaches. For example, the whitelist [12] used in [21] is quite incomplete, which may lead to inaccurate results.

Second, obfuscation may change the package names, making it harder to filter the external libraries. A most recent

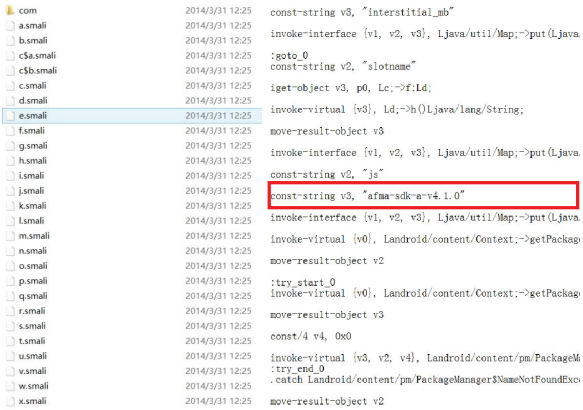

Figure 2: Library code under the root dir of SMALI.

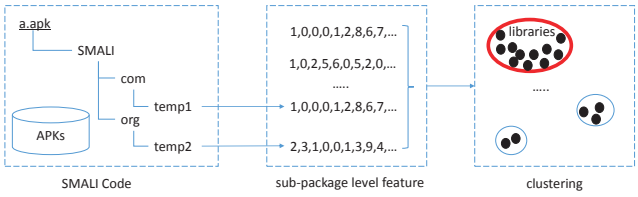

Figure 3: A clustering-based approach to filter third-party libraries.

study [42] shows that more than half of the libraries they examined have been applied different levels of obfuscation. In our study, we find many third-party libraries with package names such as "*com/a/b/*". It is impossible to filter out them using a whitelist simply by comparing their package names.

Furthermore, we also find that some libraries have no specific package names and their code is placed in the root directory of SMALI. For instance, Fig. 2 shows a library we find whose code is under the root directory of SMALI. After inspecting its SMALI code, we find that it is actually the Google AdSence library.

4.2 Clustering-based Library Filtering

We propose an accurate and automated clustering-based approach to filter external libraries, as shown in Fig. 3. We first extract the static semantic features for each non-empty sub-package in the apps. Non-empty sub-packages refers to sub-directories with code files directly under them. We only consider files in the root directory of sub-packages.

In order to enable fast comparison, the static semantic features only include the frequency of different Android API calls. Feature vectors are generated for sub-packages and then clustered into groups. We enforce *strict comparison*,

which means that only when two features are exactly the same can they be clustered. Thus, in our clustering algorithm, we first sort feature vectors according to the total number of API calls. For feature vectors with the same number of API calls, we proceed detailed comparison and cluster them if the features are exactly the same.

We assume the app dataset is large enough and each third-party library is used in many apps. As a result, the sub-packages in these libraries will be clustered in larger groups than other sub-packages, such that we could filter these libraries according to the clustering results.

We achieve the following goals with the proposed method:

- *It detects third-party libraries automatically without prior knowledge.* Therefore, it could cover more third-party libraries than whitelist-based approaches.
- *It is able to detect different versions of third-party libraries,* as long as there are enough apps using these libraries. The experiments show that we have detected different versions of Admob libraries, Apache libraries, various Android support libraries, etc.
- *It could also deal with obfuscation* because we use API calls as features, which cannot be modified during name obfuscation.

5. TWO-PHASE APP CLONE DETECTION

After filtering out the third-party libraries, the app code contains only the code representing core functionalities of each app. We use the remaining sub-packages to perform app-level clone detection. In order to perform accurate and fast app clone detection, we propose a two-phase app clone detection approach that contains *a coarse-grained detection stage and a fine-grained detection stage.*

In the coarse-grained detection stage, pairs of potentially cloned apps are selected by comparing their simple static features. In the fine-grained detection stage, we apply a code clone detection technique [60] to generate code block level features and calculate similarity scores for each app pair selected in the coarse-grained stage.

5.1 Coarse-grained Detection

5.1.1 Feature Generation

In order to achieve fast detection of suspicious cloned apps, we use simple but effective static semantic features as fingerprints, which include the call frequencies of different Android APIs. Each app is represented as a feature vector.

The intuition is that it is rare that two different apps coincidentally use exactly the same API calls, while the API calls of cloned apps should be almost the same due to their identical core functionalities. Although using this simple static fingerprint may cause false positives (e.g., two different apps have similar static features), the false negative ratio could reach almost zero. We rely on fine-grained detection in the next stage to reduce the false positives.

Other coarse-grained techniques could also be used here, such as the resource lists in [62], or hashing in [27]. We choose to use the list of API calls because it is much simpler and it can filter out dissimilar apps very fast.

5.1.2 Similarity Comparison

We use a variant of *Manhattan distance* to measure the similarity of fingerprints. For feature vectors A and B, with n kinds of features in total, their distance is represented as:

$$distance(A, B) = \frac{\sum_{i=0}^{n} |A_i - B_i|}{\sum_{i=0}^{n} (A_i + B_i)}$$

This distance is more precise than the *Jaccard distance*, which is widely used in app clone detection [63, 27]. Jaccard distance is not accurate enough because it does not consider call frequencies, which is a significant factor to measure the similarity of fingerprints. For example, it is quite different that an API is used 2 times and 100 times respectively in different apps, but the Jaccard distance cannot represent this difference and treats them equally.

If the calculated distance between two apps exceeds a certain threshold and these two apps are signed with different signatures, they will be selected as a candidate app clone pair for further fine-grained detection. Specifically, a low threshold likely leads to low false positives but high false negatives, while a high threshold introduces high false positives but low false negatives. We want to select as much app clone candidates as possible because the fine-grained detection the next stage would eliminate the false positive results. During our experiments, we empirically chose the distance 0.05 as the threshold (Section 7).

5.1.3 Pruning Strategy

Since pair-wise comparisons are computationally expensive (thousands of millions comparisons), we introduce some optimization strategies.

If two apps are an app clone pair, most of their attributes will not differ much. Typically, these attributes include the total number of API calls and the total kinds of API calls, which are the meta-data of the feature vector. These attributes will not be affected much with minor modifications, and thus they are typically stable. If two apps are "very different" in the meta-data, we will stop compare the feature vectors and mark them as dissimilar. The two numbers should not differ by a large ratio, such that 100 and 50 will be regarded as "very different". During our experiments, if the attributes of two feature vectors differ by more than 20%, we will stop compare them.

5.2 Fine-grained Detection

For the selected potentially cloned apps during the coarse-grained phase, we further exploit more detailed features to calculate their similarity. This step is based on an existing *counting-based code clone detection* approach called Boreas [59, 60], which has been independently implemented and confirmed as both accurate and scalable [22].

The basic idea is that we match the variables, rather than the sequences or structures of code segments. The similarity of two code segments[3] is decided by the proportion of variables that could be matched based on their *feature matrices*, which are formed by counting the number of times each variable occurs in different contexts. Accordingly, the similarity of two apps is measured by the proportion of similar code segments.

We have successfully applied Boreas to detect repackaged Android app clones in our previous work [55]. Although it is able to identify repackaged apps accurately, because it requires the extraction of detailed features and perform pair-wise comparison on all apps, the performance is unacceptable for large-scale detection. In this paper, we modify it

[3]We choose code segments separated by natural punctuation marks as the basic clone granularity.

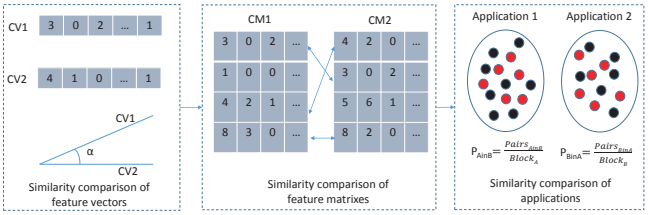

Figure 4: The workflow of fine-grained similarity comparison

and use it as the technique behind the fine-grained detection stage, instead of applying it to all apps directly.

5.2.1 Feature Generation

As mentioned above, we generate feature matrices based on the number of occurrences of all variables counted in different contexts. We use *Counting Environments* (CEs for short) to describe the contexts.

The CEs are divided into three levels, each level provides a more concrete and distinct description for chosen variables:

- *Naive counting*, which includes the simplest counting information: the number of times each variable is used and defined.
- *In-statement counting*, which includes more high-level information involving only a single statement, for example, the number of times each variable occurs in an if-predicates, the number of times each variable is added or subtracted, or the number of times each variable occurs as an array subscript.
- *Inter-statement counting*, which involves the environments involving multiple statements, such as the nested loop-level of variables. It includes the number of times each variable occurs in a first-level loop, a second level loop and a deeper level loop.

For each variable, we generate an m-dimensional *Characteristic Vector* (CV for short) using m CEs, where the i-th dimension of the CV is the number of occurrence of the variable in the i-th CE. For each code segment, we compute CVs for all variables. Then we can obtain a $n \times m$ *Characteristic Matrix* (CM for short). A CM represents the abstraction for a code segment. For each app, we compute the CM for each code segment. Then we can obtain a series of CMs, which we treat as the code segment level features of this app.

5.2.2 Similarity Comparison

As shown in Fig. 4, we perform similarity comparison of feature vectors, code segments and apps, respectively. The similarity of two apps is measured by the proportion of their similar code segments. The similarity of code segments is determined by the corresponding feature matrices, while the comparison of feature matrices is decided by the similarity comparison of their feature vectors.

Similarity of Feature Vectors. We use Cosine similarity to compare CVs. Because CVs represent the patterns of variables, the similarity of two variables can be computed by the cosine of vectors in high dimensional spaces.

For two vectors a and b with the angle α between them, their cosine similarity is defined as:

$$CosSim = \cos(\alpha) = \frac{a \cdot b}{||a||||b||} = \frac{\sum_{i=1}^{m} a_i \times b_i}{\sqrt{\sum_{i=1}^{m} a_i^2} \times \sqrt{\sum_{i=1}^{m} b_i^2}}$$

Similarity of Code Segments. The similarity of two code segments is defined as the similarity of their CMs, which is related to the matching of their variables.

Variables are sorted according to their frequencies. We match each variable a of block A to those variables of block B whose ranks are close to the rank of a. Duplicated matches are allowed, that is, although every variable of block A must match exactly one variable of block B, there are no such restrictions on the variables of block B. It greatly simplifies comparison process: we only need to search a small range of variables for each variable of block A, choose the most similar one as the similarity value for each variable, then compute the product of these similarity values.

Similarity of Apps. The similarity between two apps is calculated as the proportion of similar code blocks in them. For each app pair A and B, we calculate two similarity scores: $Sim_{A(B)}$ and $Sim_{B(A)}$. A higher similarity score means that a large portion of one app can be found in the other, providing the evidence of app cloning. We identify two apps as clones when at least one of these two similarity scores are over the threshold ($max(Sim_{A(B)}, Sim_{A(B)}) >= threshold$). During our experiment, we empirically chose the threshold as 85%(Section 7).

6. IMPLEMENTATION

We implement WuKong in a prototype system, which includes roughly 6,912 lines of C++ code, 3,300 lines of python code and 780 lines of shell script code.

The implementation of WuKong involves the following steps:

- In the pre-processing stage, we disassemble APKs to SMALI code using Apktool [5]. Keytool [11] is used to extract the developer's signature. We extract static semantic features for each sub-package and cluster them into groups by strict comparison. The groups whose size are greater than the threshold are regarded as third-party library related code.
- In the coarse-grained detection phase, we calculate the feature fingerprint for each app by accumulating the static features of non-library sub-packages. Then, we perform pair-wise comparisons to calculate the distance between apps. If the similarity score exceeds the threshold, the corresponding app pair will be classified as app clone candidates.
- In the fine-grained detection phase, for each selected suspicious app clone pair, we decompile their core functional code to Java code using Dex2Jar [7] and JD-Core-Java [10]. We have modified JD-Core-Java in order to execute in batch through command-line. We compute the feature matrices for each app, and calculate the similarity scores for each suspicious app clone pair detected in the coarse-grained detection stage. If the similarity score exceeds the specified threshold, they will be considered as a cloned pair, which represents the final detection result.

7. EVALUATION

We evaluate WuKong with apps downloaded from five popular third-party Android markets. Our experiments are

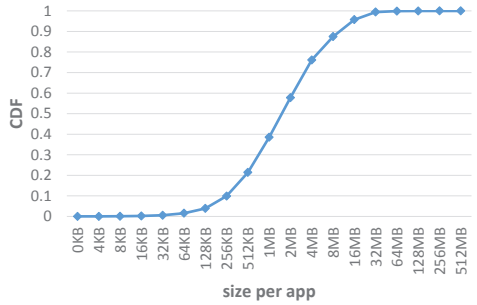

Figure 5: The distribution of the sizes of APK files

Table 1: Experiment Dataset

Market	Number of Apps	Percentage of Dataset
anzhi [3]	14,047	13.3%
eoe [8]	40,134	38.1%
gfan [9]	13,672	13.0%
baidu [6]	16,613	15.8%
myapp [13]	20,833	19.8%
total	105,299	100%

conducted on a Lenovo Thinkcenter with CORE i7 3.40GHz CPU and 4GB memory.

7.1 Dataset Statistics

We collected 105,299 Android apps from five different markets[4]. The apks and decompiled resources occupy nearly 4TB storage space. The distribution of collected apps from different markets is shown in Table 1. Fig. 5 shows the distribution of the sizes of APK files. The majority of the sizes are between 512KB and 16MB, as the sizes between 1MB and 8MB account for more than 50% of the apps. The sizes of more than 60% apps exceed 1MB.

7.2 Pre-processing

We generate feature fingerprints for each sub-package of apps. There are total 4,406,128 non-empty sub-packages.

Fig. 6 shows the distribution of the number of API calls per sub-package. The number of API calls vary greatly for different sub-packages. Large sub-packages contain more than 60,000 API calls, while small sub-packages only contain less than 10 API calls. Each sub-package has 309 API calls on average.

7.2.1 *Third-party Library Filtering*

We cluster the sub-packages into groups according to their feature vectors. It is worthy to note that small sub-packages contain much fewer API calls, which would impact the results of clustering. Thus we exclude small sub-packages which contain less than 10 different APIs during clustering. The threshold is determined according to the clustering results. Clusters with sizes larger than the threshold will be filtered as external libraries.

Sub-package Level Clustering. After clustering, the distribution of clusters with different sizes is shown in Fig. 7. We can see that the clusters with size 2 or 3 account for the most number of clusters and number of sub-packages.

[4]We do not use apps from the official Google Play market because we were unable to download a large number of apps from Google Play, because we cannot access Google Play directly from China during the same period.

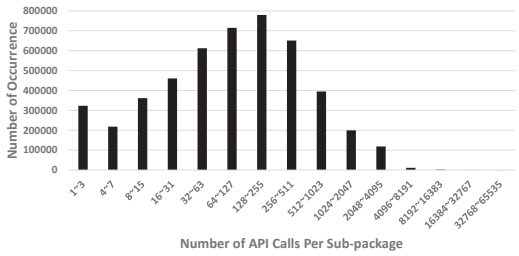

Figure 6: The distribution of the number of API calls per sub-package

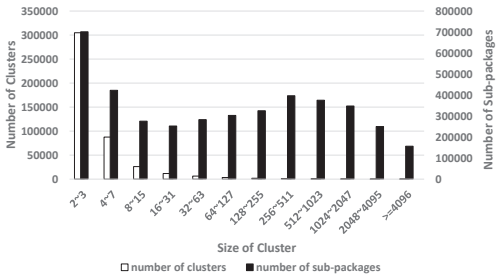

Figure 7: The distribution of the size of cluster (sub-package level clustering)

With the increase of cluster size, the number of clusters decrease greatly. But the total number of sub-packages in these clusters still account for a large portion in the clustered results.

Table 2 shows the top 12 clusters with their representative package names. Note that although the sub-packages are clustered into the same group, their package names may be not identical. In that case we randomly choose a sub-package name (as shown in the table).

We could see that the top five clusters are sub-packages related to Android support libraries. Many of these top clusters are not ad libraries, thus not listed in most whitelist approaches. An interesting point is that sub-packages in different clusters may use the same package names. Such as in Table 2, there are two clusters with the representative name "*/smali/android/support/v4/net*". The reason is that we use strict comparison in sub-package level clustering, only when the features are exactly the same can they be grouped together. A minor difference in feature vectors will lead to that they are clustered to different groups. As a result, we could detect different versions of libraries.

The results also show that many third-party libraries are obfuscated. Such as the sub-packages with package name "*/smali/com/gpworkstui/kz/a/c*" in Table 2. These obfuscated libraries cannot be filtered with whitelist approaches.

The threshold of third-party libraries. To filter third-party libraries, we need determine a threshold after clustering. In our experiments, we combine two ways to determine the threshold.

First, we download 200 apps containing more than 60 different identified third-party libraries. We manually check their disassembled SMALI code, and label the sub-packages of the corresponding libraries. We cluster the sub-packages of these 200 apps with the 105,299 apps downloaded previ-

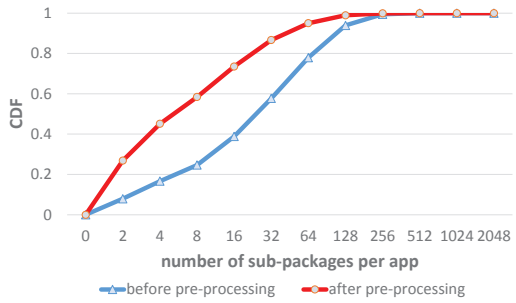

Figure 8: The distribution of the number of non-empty sub-package per app before/after pre-processing

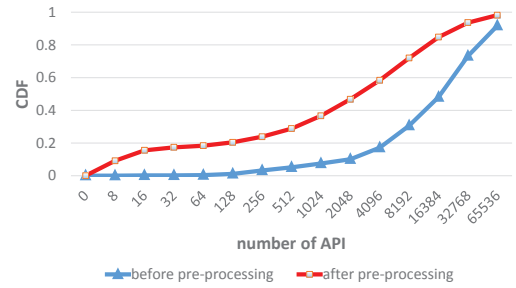

Figure 9: The distribution of the number of API calls per app before/after pre-processing

ously. In the results of the detected sub-packages, we could see the distribution of these known libraries.

Second, for different cluster sizes (as shown in Fig. 7), we randomly choose 5 clusters to manually check their SMALI code and label them as "library", "core code", or "cannot decide". For some sub-packages, it is easy to decide according to their package names. While for others, we need look for some specific statements in their code and use Google search to help us make decisions. We checked about 60 clusters with sizes ranged from 2 to 5000.

Combing the experiment results of these two approaches, we choose 32 as the threshold to filter libraries. Clusters with sizes larger than 32 will be labeled as third-party libraries. With this threshold, we are able to find more than 600 different third-party libraries (each of them appearing in more than 32 different apps), which is much larger than the whitelist approaches in previous studies.

Refinement. As stated above, there are some small sub-packages containing fewer than 10 different API calls. We have not clustered these sub-packages with others to avoid false positives, because there is a higher possibility for small sub-packages to be coincidentally the same with each other. We collect the package names of identified third-party library according to the clustering results and known library lists, and use these names to filter some small sub-packages. Although we still could not filter all the third-party related small sub-packages, they have nearly negligible effects on the results of app clone detection.

Results. Fig. 8 shows the distribution of the number of sub-package per app before and after pre-processing. Before pre-processing, more than 70% of the apps contain 8 to 127 non-empty sub-packages and every app has 41.8 non-empty sub-packages on average. After pre-processing, most apps contain 1 to 32 sub-packages and each app contains 15.8 sub-packages on average. Besides, we also compare the number of API calls before and after pre-processing, as shown in Fig. 9. The number of API calls decrease greatly. The overall results of pre-processing are shown in Table 3. More than 60% of the non-empty sub-packages and more than 57% of the API calls are filtered.

Table 2: The Top 12 clusters and their representative package names

Cluster Size	Package Name
19,880	android/support/v4/accessibilityservice
13.494	android/support/v4/view/accessibility
10.568	android/support/v4/net
8.985	android/support/v4/widget
8.161	android/support/v4/util
7.361	com/google/ads/mediation
6.923	android/support/v4/net
6.865	org/apache/http/entity/mime
6.413	org/apache/http/entity/mime/content
6.287	com/gpworkstui/kz/a/c
5.477	com/android/vending/billing
5.381	com/google/gson/reflect

Table 3: The results of app pre-processing

	before filtering	after filtering	percentage filtered
sub-packages	4,406,128	1,665,970	62.2%
API calls	1,363,293,287	575,691,934	57.8%

7.3 Coarse-grained Detection

7.3.1 *Feature Generation*

We calculate the feature vectors for core functionalities of each app after pre-processing. As shown in Figure 9, more than 60% apps contain more than 1024 API calls after pre-processing, while about 17% of the apps contain only fewer than 16 API calls.

7.3.2 *Determining the Threshold*

In the coarse-grained detection process, we aim to improve the comparison efficiency while obtaining sufficient accuracy. To this end, we choose 1000 samples and use a series of distance thresholds to measure their accuracy.

We first get the ground truth of app clones by manually checking, installing and comparing these 1000 apps. Then we use the coarse-grained detection to compare these 1000 samples pair-wisely (the total number of app pairs is 499,500). We calculate the true positives and false positives under different threshold by examining each reported pair. We find that 0.05 can be used as the optimal distance to achieve the most true positives. Although there are about 10% false positives under the distance of 0.05, we still want to achieve more true positives because the subsequent fine-grained detection phase could reduce the false positives detected in coarse-grained detection.

7.3.3 *Coarse-grained Detection Results*

After the threshold is chosen, we apply it to detect suspicious cloned apps from our dataset. The total number of app pairs is about 5 billion. Luckily, our pruning strategy could reduce a large portion of the app pairs to accelerate the process.

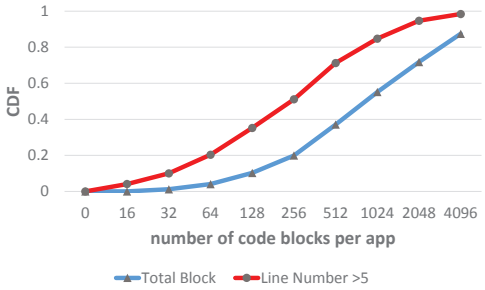

Figure 10: The distribution of number of code blocks per app

After this step, we detected 93,122 suspicious app clone pairs, which include 14,702 apps. The result is shown in Table 4. Although the suspicious candidates comprise about 14% of the total apps, we are able to narrow down the app pairs by almost five orders of magnitude, which greatly reduces the burden of fine-grained detection.

7.4 Fine-grained Detection

7.4.1 Feature Generation

We calculate the CMs for each selected app. The distribution of apps with different number of code blocks is shown in Fig. 10. Note that because small blocks (lines of code <=5) usually contain much less information, it makes no sense to include them in app clone detection. Therefore, we filter small code blocks that contain fewer than 5 lines of code first. The distribution of apps with different number of code blocks after filtering is also shown in Fig. 10. We could see that the number of code blocks decrease greatly, which improves the detection efficiency of this phase.

7.4.2 Determining the Threshold

We randomly choose 500 samples and use the fine-grained approach to compare them pair-wisely. We manually check, install and compare these 500 apps to get the ground truth of app clones first. Then we use the fine-grained approach to compare them pair-wisely (124,750 app pairs).

We use a series of similarity thresholds to measure their accuracy. Based on the false positives and false negatives under different thresholds, we choose 85% as the optimal threshold.

7.4.3 Fine-grained Detection Results

We compare the selected 93,112 suspicious app pairs after coarse-grained detection. Apps with similarity scores exceeding the specified threshold will be considered as app clone pairs. After fine-grained detection, we detected 80,439 app clone pairs, which include 12,922 apps. The results are shown in Table 4. About 12% of the apps in our dataset are detected as cloned apps in the identified app clone pairs.

We further study the cross-market and inside-market app clone situations, which are shown in Fig. 11. An interesting finding is that *besides cross-market app clone pairs, there exist many clone pairs even within the same market.* For example, *eoe* contains 4,368 apps that are clones within the market, which represents more than 10% of the total apps we studied from this market. The results show that, *it is very important for an app store/market to apply app clone detection techniques to identify and remove these cloned apps from their market.*

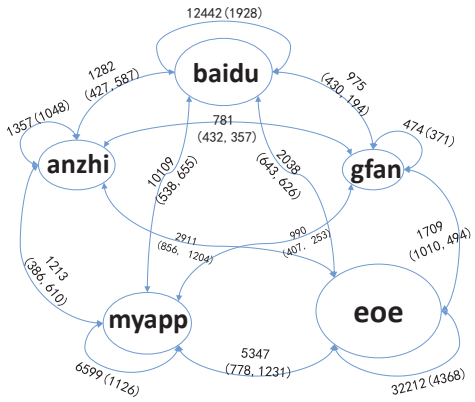

Figure 11: Cross-market and inside-market app clones. (*Each node corresponds to a market. The size of a market node is proportional to the number of apps from the market. The number beside an edge shows how many app clone pairs are cross the two markets. The numbers in the brackets refer to the number of involved apps in each of the two markets.*)

7.5 The Gap Between Two Phases

7.5.1 False Positive of First Phase

From the detection results, we could see that about 12% of the apps and 13.6% of app pairs detected in coarse-grained detection are not considered as app clone pairs after the fine-grained detection stage.

We randomly choose 100 of these app pairs for further analysis. We manually check, install and compare these apps, and we find all of them are not app clones, which means that they are the false positives of first phase.

7.5.2 The Reasons Leading to the Gap

We identify the following reasons leading to the gap between two phases:

(1) A large portion of these app pairs are small apps, which contain very few APIs (<10) and code blocks. We even find some apps containing only 4 or 5 API calls, thus it is quite possible that they are detected as clone pairs in the coarse-grained detection stage.

(2) Many of these app pairs belong to the same categories, such as wallpaper apps and clock apps. They have similar simple functions (such as choosing a wallpaper), thus the number of API calls in these apps are similar. Therefore, they are selected as app clone pair incorrectly (in the coarse-grained stage). But in the fine-grained detection, their similarity score is much lower because the difference in their code blocks. For example, we find one app with package name "*lf.live.hjfceby10.apk*" which is detected as in a clone pair with another app with package name "*com.mobi.screensaver.model6.apk*" in the coarse-grained detection stage. However, the first app contains 647 blocks, while the second app only contains 46 blocks. They are both wallpaper apps, but the first app contains many other code with no API calls.

7.6 Accuracy

We use two ways to measure the false positives of our two-phase approach.

Table 4: The results of two-phase app clone detection

Market	total # of apps	coarse-grained results	coarse-grained / total apps	fine-grained results	fine-grained/ coarse-grained	fine-grained/ total
anzhi	14,047	2,306	16.4%	2,022	87.8%	14.4%
eoe	40,134	6,307	15.7%	5,565	88.2%	13.9%
gfan	13,672	1,460	10.7%	1,068	73.2%	7.8%
baidu	16,613	2,654	16.0%	2,473	93.2%	14.9%
myapp	20,833	1,982	9.5%	1,799	90.8%	8.6%
total	105,299	14,702	14.0%	12,922	87.9%	12.2%
app pairs	5,543,887,051	93,112	0.00168%	80,439	86.4%	0.00145%

(1) First, we use Androguard [2] to help measure the false positives. Androguard is a reverse engineering and static analyze tool for Android apps, which provides the feature of measuring the similarity of Android apps. To the best of our knowledge, it is the only repackaging detection tool that is available for use. It calculates the similarity score of apps by identifying method relevant metrics. We randomly select 2,000 of the 80,439 detected app clone pairs and apply Androguard on these apps. Note that we filter the third-party libraries first to keep the same experiment environment with our approach. With the threshold of 85%, Androguard could detect 1,931 app clone pairs out of the 2,000 app pairs. We randomly select 500 of the 1,931 app clone pairs detected by both our approach and Androguard. We manually check these app pairs by inspecting their disassembled code and installing them on smartphones. We do not find any false positives. In the same way, we manually check the 69 app clone pairs that detected by our approach but can not be detected by Androguard. We find that they are indeed repackaged apps. It shows that *our approach has no false positives in this experiment and our approach is more accurate than Androguard.*

(2) At the beginning of the evaluation, we add 1000 labeled apps to the dataset. Among the 1000 apps, there are 58 manually verified app clone pairs (116 apps) downloaded from various markets with similar names or descriptions. Besides, there are 100 artificially generated app clone pairs (200 apps). We use various ways to generate these app clone pairs, including using Proguard [14] to obfuscate the apps, using APIMonitor [4] to insert some monitor code and reordering the method manually. The remaining 684 apps are totally different. We pre-process and detect these apps together with the 105,299 apps. We check the detection results to analyze the results containing these 1000 apps. We are able to detect all these 158 app clone pairs successfully. Meanwhile, we also find *no false positives* among these 1000 apps (499,500 app pairs).

We do not attempt to measure the false negative because there is no feasible way to find the ground truth for the 105,299 apps in our dataset. As a matter of fact, none of the previous app clone detection approaches [64, 27, 62, 54, 63, 61, 55, 21, 25] have measured the false negative rate.

7.7 Scalability

We analyze the performance of different detecting stages respectively. In the pre-processing stage, after disassembling the apps, we extract the static semantic features for each sub-package. It takes about 28 hours to extract and store the features for these 4,406,128 sub-packages. Then we cluster these sub-packages into groups to filter third-party libraries, and it takes less than 60 minutes. In the coarse-grained stage, it takes about 4 hours to detect suspicious app clones, including 2 hours to generate the features for each app and 2 hours to compare these apps pair-wisely. In the fine-grained stage, it takes about 10 hours to generate the code segment level features for each selected apps. For the selected suspicious app clone pairs, it take about 2.5 hours to calculate the final similarity scores.

In comparison, comparing each app pair during coarse-grained detection takes roughly 0.0000013 seconds on average, while it takes 0.097 seconds to compare each app pair in the final fine-grained phase. Although the fine-grained phase takes much longer to compare each pair, it compares much fewer app pairs because the coarse-grained phase has successfully filtered most of those dissimilar app pairs.

Note that we run all the phases on a single thread, which leaves room for further improvement to take advantage of multiple cores for speed-up.

8. DISCUSSIONS

In this section, we examine possible limitations of WuKong and potential future improvements.

Code Obfuscation. ProGuard is the most widely used obfuscation tool that has been integrated into the Android build system. Proguard obfuscates code by renaming classes, fields, and methods with semantically obscure names. WuKong could handle this kind of obfuscation well. However, hackers could use some complex obfuscation algorithms such as Class Encrypter [29] and API obfuscation to evade our detection, which need to be further investigated.

App Clones VS. Original Apps. Although we can detect app clone pairs from a very large number of apps, for a given clone pair, it is hard to decide which one is the original app. Previous work [26, 63] have proposed some heuristic solutions, such as checking the submission time of the app, the popularity of the app (download times), the size of APK, or the proportion of non-primary code. However, none of these solutions are perfectly sound and it is easy for hackers to evade detection.

App Containment. It is possible that a cloned app contains several small apps or the cloned app has far more code than the original app. In this case, our approach may not detect it as an app clone due to significant difference in their features. It is hard to determine whether they are actually app clones or not, because they possess many different functionalities.

We might be able to detect this kind of clones with minor modification to WuKong, such as considering the include-relationship when comparing app signatures. However, it will greatly increase the complexity of comparison. Meanwhile, false positives may increase, because the features of small apps are likely to be contained in many big apps.

Usage Scenario. One typical usage scenario is applying mobile app clone detection to an app market, where each newly submitted app will be compared to all the existing apps to check whether it is a cloned app. WuKong can be easily modified to adapt to such cases. Since we only need to compare one app to all the existing apps, we should be able to finish the comparison within seconds even if there are more than 1 million apps (such as in Google Play). We will investigate this issue in more details in future work.

9. RELATED WORK

App Clone Detection. There are many recent work studying how to detect Android app clones.

Several earlier approaches use simple features such as hashing. DroidMOSS [64] collects the syntactic instruction sequences to generate features. Fuzzy hashing [34] is used to generate the fingerprints. Juxtapp [27] leverages feature hashing [58] to detect code reuse in Android apps.

FSquaDRA [62] and PlayDrone [54] use resource signatures to perform fast detection. Androguard [2] calculates the similarity score of apps by identifying method relevant metrics. Zhou et al. [63] proposed a module decompiling technique to partition an app's code into primary and non-primary modules. Then semantic features are extracted from the primary modules to detect "piggybacked" apps.

ViewDroid [61] leverages user interface based birthmark to detect app clones, which is resilient to code obfuscation. However, for the detection of apps with few views, the false positive ratio is high. CLAN [44] detects similar Java apps using Java API calls, which is similar with our coarse-grained approach. In general, these coarse-grained techniques using simple features are fast, while providing relatively less accurate results.

There are also detection techniques using more complicated features. For example, DNADroid [24] detects cloned apps by comparing the PDGs between methods in candidate apps. Wang et al. [55] leveraged counting-based code clone detection techniques, which also faces scalability difficulties because their complexity.

AdDarwin [25] splits PDGs into connected components and extract semantics vector for each component. Semantic vectors are calculated by counting the occurrence frequency of specific types (e.g., binary operation type). AdDarwin also uses semantics vectors to detect external libraries. However, semantics block level clustering would introduce high false positive rate [21].

Chen et al. [21] use the geometry characteristic (centroid) of dependency graphs to measure the similarity between methods of two apps. Their solution has demonstrated to be both scalable and accurate. However, they also use a whitelist to filter third-party libraries, which could lead to inaccurate results.

As a matter of fact, our proposed technique for third-party library filtering can be incorporated to many existing techniques such as [21] to improve their performance.

Code Clone Detection. A more general related research area is code clone detection, which have been studied extensively for dozens of years. Text-based techniques [37, 47, 16] use little or no transformation on the source code before the actual comparison, and in most cases raw source code is used directly in the clone detection process.

Token-based techniques [17, 18, 32, 38] begin by transforming source code into a sequence of "tokens" using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones.

Counting-based techniques [59, 60] could improve the accuracy of token-based clone detection and are effective in some apps such as detecting programming bugs and plagiarisms. The fine-grained detection in WuKong is based on a counting-based code clone detection technique.

Syntactic approaches [19, 20, 48, 50, 23] use a parser to convert source programs into parse trees or abstract syntax trees, which can then be processed using either tree matching or structural metrics to find clones. Lee et al. [36] proposed a multi-dimensional token-level indexing structure using an R* tree on Deckard's vectors [31]. Semantics approaches [35, 28] use static program analysis to provide more precise information than simply syntactic similarity. Kim et al. [33] proposed a symbolic-based approach to identify semantically equivalent procedures. However, these approaches are generally slower and not scalable enough.

Software Plagiarism Detection. Another related research area is software plagiarism detection. *Software birthmark* is used to detect software plagiarism, which is a unique characteristic that a program inherently possesses that can be used to determine the identity of a software.

Software birthmarks could be classified into static birthmark and dynamic birthmark. Tamada [51] proposed four types of static birthmark: constant values in field variables birthmark, sequence of method calls birthmark, inheritance structure birthmark and used classes birthmark. GPLAG [43] leveraged PDG-based birthmark. Lim et al. [39] used stack pattern based birthmark. Myles et al. [45] statically analyzed executables and proposed op-code level k-gram based static birthmark. Lim et al. [40] proposed n-gram flow-path birthmark. Static birthmarks are vulnerable to obfuscation techniques, such as instruction reordering.

Dynamic software birthmarks include dynamic API based birthmarks [53, 52, 49], whole program path birthmark [46], system call based birthmark [56, 57] and core value based birthmark [30]. They need dynamically analyze the program, thus not suitable for large scale plagiarism detection, such as cross-market app clone detection.

10. CONCLUSION

This paper presents WuKong, a new accurate and scalable approach to detect Android app clones. Our proposed techniques include a novel clustering-based approach to detect third-party libraries efficiently and accurately without prior knowledge, which introduces significant benefits compared to previously used whitelist approaches. We also introduce a two-phase approach to detect app clones, which combines the scalability of coarse-grained detection and the accuracy of fine-grained detection mechanisms. Experiments on over 100,000 Android apps show that WuKong is able to detect app clones within several hours of comparison, with no false positives found in our evaluation.

Acknowledgment

This work is partly supported by the High-Tech Research and Development Program of China (863) under Grant No.2015AA01A203, the National Basic Research Program of China (973) under Grant No. 2011CB302604, and the National Natural Science Foundation of China under Grant No.61421091, 61370020, 61103026.

11. REFERENCES

[1] Daily Android activations grow to 1.5 million, Google Play surpasses 50 billion downloads. http://bgr.com/2013/07/20/android-activations-app-downloads/, 2013.

[2] Androguard. https://code.google.com/p/androguard/, 2014.

[3] Anzhi market. http://www.anzhi.com/, 2014.

[4] Apimonitor. https://code.google.com/p/droidbox/wiki/APIMonitor, 2014.

[5] Apktool. https://code.google.com/p/android-apktool/, 2014.

[6] Baidu market. http://shouji.baidu.com/, 2014.

[7] Dex2jar. https://code.google.com/p/dex2jar, 2014.

[8] Eoe market. http://www.eoemarket.com/, 2014.

[9] Gfan market. http://apk.gfan.com/, 2014.

[10] Jd-Core-Java. https://github.com/nviennot/jd-core-java, 2014.

[11] Keytool. http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html, 2014.

[12] A list of shared libraries and Ad libraries used in Android apps. http://sites.psu.edu/kaichen/2014/02/20/a-list-of-shared-libraries-and-ad-libraries-used-in-android-apps/, 2014.

[13] Myapp market. http://android.myapp.com/, 2014.

[14] Proguard. https://proguard.sourceforge.net/, 2014.

[15] Smali: An assembler/disassembler for Android's dex format. https://code.google.com/p/smali, 2014.

[16] B. S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, 1992.

[17] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86–95, 1995.

[18] B. S. Baker. Parameterized pattern matching: algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996.

[19] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L.Bier. Clone detection using abstract syntax trees. In *Proceedings of the 1998 International Conference on Software Maintenance (ICSM)*, 1998.

[20] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. In *SYRCOSE*, 2008.

[21] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, 2014.

[22] X. Chen, A. Y. Wang, and E. D. Tempero. A replication and reproduction of code clone detection studies. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference (ACSC)*, pages 105–114, 2014.

[23] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. A tree kernel based approach for clone detection. In *Proceedings of the 2010 International Conference on Software Maintenance (ICSM '10)*, pages 1–5, 2010.

[24] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: detecting cloned applications on Android markets. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS '12)*, 2012.

[25] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar Android applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS '13)*, 2013.

[26] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. AdRob: examining the landscape and impact of Android application plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*, pages 431–444, 2013.

[27] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: a scalable system for detecting code reuse among Android applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA '12)*, 2012.

[28] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: a PDG-based approach. In *WCRE*, pages 3 –12, 2011.

[29] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithm. In *Proceedings of the 6th International Conference on Trust and Trustworthy Computing*, 2013.

[30] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 756–765, 2011.

[31] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 96–105, 2007.

[32] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transaction on Software Engineering*, 28(7):654–670, 2002.

[33] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 301–310, 2011.

[34] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digit. Investig.*, 3:91–97, Sept. 2006.

[35] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[36] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pages 167–176, 2010.

[37] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pages 140–141, 2005.

[38] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transaction on Software Engineering*, 32(3):176–192, 2006.

[39] H.-i. Lim, H. Park, S. Choi, and T. Han. Detecting theft of Java applications via a static birthmark based on weighted stack patterns. *IEICE - Trans. Inf. Syst.*, E91-D(9):2323–2332, 2008.

[40] H.-i. Lim, H. Park, S. Choi, and T. Han. A method for detecting the theft of Java programs through analysis of the control flow information. *Inf. Softw. Technol.*, 51(9):1338–1350, 2009.

[41] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk. Revisiting Android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 242–251. ACM, 2014.

[42] B. Liu, B. Liu, H. Jin, and R. View. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the The 13th International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, 2015.

[43] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881, 2006.

[44] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 364–374, 2012.

[45] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 314–318.

[46] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In *Information security*, pages 404–415, 2004.

[47] C. K. Roy and J. R. Cordy. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[48] P. Schugerl. Scalable clone detection using description logic. In *IWSC '11*, pages 47–53, 2011.

[49] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for Java. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pages 274–283, 2007.

[50] G. Selim, K. C. Foo, and Y. Zou. Enhancing source-based clone detection using intermediate representation. In *WCRE*, pages 227 –236, 2010.

[51] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of Java programs. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 569–575, 2004.

[52] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of dynamic software birthmarks based on API calls. Technical report, Nara Institute of Science and Technology, 2007.

[53] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.-I. Matsumoto. Dynamic software birthmarks to detect the theft of Windows applications. In *Proceedings of the International Symposium on Future Software Technology (ISFST '04)*, 2004.

[54] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*, pages 221–233, 2014.

[55] H. Wang, Z. Wang, Y. Guo, and X. Chen. Detecting repackaged Android applications based on code clone detection technique. In *SCIENCE CHINA Information Sciences*, volume 44(1), pages 142–157, 2014.

[56] X. Wang, Y. chan Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 149–158, 2009.

[57] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 280–290, 2009.

[58] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*, pages 1113–1120, 2009.

[59] Y. Yuan and Y. Guo. CMCD: count matrix based code clone detection. In *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC '11)*, pages 250–257, 2011.

[60] Y. Yuan and Y. Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, pages 286–289, 2012.

[61] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '14)*, 2014.

[62] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser. FSquaDRA: fast detection of repackaged applications. In *Data and Applications Security and Privacy XXVIII*, volume 8566 of *Lecture Notes in Computer Science*, pages 130–145. 2014.

[63] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, pages 185–196, 2013.

[64] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY '12)*, 2012.

[65] Y. Zhou and X. Jiang. Dissecting Android malware: characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*, pages 95–109, 2012.