Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets

Kai Chen[†,‡], Peng Liu[‡], Yingjun Zhang[§]
[‡]College of IST, Penn State University, U.S.A.
[†]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, China
[§]Institute of Software, Chinese Academy of Sciences, China
chenkai010@gmail.com, pliu@ist.psu.edu, yjzhang@tca.iscas.ac.cn

ABSTRACT

Besides traditional problems such as potential bugs, (smartphone) application clones on Android markets bring new threats. That is, attackers clone the code from legitimate Android applications, assemble it with malicious code or advertisements, and publish these "purpose-added" app clones on the same or other markets for benefits. Three inherent and unique characteristics make app clones difficult to detect by existing techniques: a billion opcode problem caused by cross-market publishing, gap between code clones and app clones, and prevalent Type 2 and Type 3 clones.

Existing techniques achieve either accuracy or scalability, but not both. To achieve both goals, we use a geometry characteristic, called *centroid*, of dependency graphs to measure the similarity between methods (code fragments) in two apps. Then we synthesize the method-level similarities and draw a Y/N conclusion on app (core functionality) cloning. The observed "*centroid effect*" and the inherent "*monotonicity*" property enable our approach to achieve both high accuracy and scalability. We implemented the app clone detection system and evaluated it on five whole Android markets (including 150,145 apps, 203 million methods and 26 billion opcodes). It takes less than one hour to perform cross-market app clone detection on the five markets after generating centroids only once.

Categories and Subject Descriptors

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

General Terms

Languages, Algorithms, Experimentation, Security

Keywords

Software analysis, Android, clone detection, centroid

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

1. INTRODUCTION

Code clones, or similar fragments of source code, bring unwanted problems such as inconsistencies [21], potential bugs [38] and code smells [14]. Today, with the rapidly increasing use of smartphones, code clones in smartphone applications (*apps* for short) bring several kinds of new threats. Specially, some attackers clone the code from legitimate Android apps and pack/assemble it with "purpose-added" functionalities or modifications after reverse-engineering those apps [43, 35]. This new kind of code clone is referred to as *app clones* or *repackaging* [53]. After cloning an app, attackers would upload it to the same market (e.g., Google Play [16]) or other markets.

App clones bring severe problems. (a) Smartphone malware prefers to use app clones as "carriers" for propagation. Zhou *et al.* [54] found 1083 of 1260 (or 86.0%) malware samples are app clones with malicious payloads. (b) Legitimate developers lost their advertising revenue and users to app clones. According to a recent study [15], 14% of the advertising revenue and 10% of the user base for a developer are diverted to app clones on average.

1.1 Unique Characteristics of App Clones

App clones meet two essential criterions. (1) A large portion of the core functionalities of one app are cloned in another app. To separate app clones from shared libraries, we do not view common frameworks and third-party libraries as part of core functionalities. (2) App clones are developed by different authors/companies. Based on the two criterions, we find that app clones on Android markets have the following unique characteristics.

Characteristic 1: A billion opcode problem. Since two app clones could appear on different markets, cross-market analysis is necessary. Given multiple markets, the app clone detection problem is a billion *opcode* problem. Since most developers do not release the source code, we use the number of *opcodes* to measure the amount of code to analyze. An opcode is a part of a bytecode statement, which determines different kinds of VM operations. Hanna *et al.* [17] analyzed 30,000 apps from Google Play (the official Android market) and found that the total number of opcodes in all apps is approximately 1.45 billion.

Characteristic 2: Gap between code fragment clones and app clones. Traditional clone detection is conducted inside a big software project (e.g., Apache) to identify similar code fragments, not similar apps. In contrast, detecting app clones needs to pairwisely compare apps in multiple markets.

A smartphone app is a set of code fragments. Although code fragment clone detection is still very useful, two apps containing code fragment clones are not necessarily app clones due to two reasons. R1) The two apps could use common third-party libraries. R2) Unless the core functionalities of two apps are cloned, we cannot say that they are clones.

Characteristic 3: Type 2 and Type 3 clones are prevalent on Android markets. Based on several studies [9, 53], many app clones on Android markets are Type 2 and/or Type 3 clones (The four types of clones are defined in the footnote[1]). In contrast, traditional code clones are not always Type 2 and Type 3. In fact, many traditional clones are Type 1. Type 4 clones rarely exist in app clones. This is mainly because attackers usually do not bother to understand and perform advanced transformation on the bytecode of legitimate apps.

1.2 Prior Work and Motivation

Although considerable research has been conducted on clone detection, unfortunately, existing techniques are not suitable for detecting app clones on Android Markets. We compare these techniques in Table 1.

String-based [5], token-based [22, 28] and AST (Abstract Syntax Tree)-based [7, 6, 20, 26] approaches have shown their scalability to handle millions of lines of code. However, they generate too many false negatives at handling Type 2 and/or Type 3 clones (Characteristic 3).

PDG (Program Dependence Graph)-based approaches [31, 13] capture the control flow and data dependencies between the code statements inside code fragments. They can effectively detect Type 2 and Type 3 clones. However, PDG comparison by graph isomorphism analysis is **not scalable**, which cannot handle billions of opcodes in multiple markets (Characteristic 1).

Three recent studies try to handle this problem. Gabel *et al.* reduce PDG comparisons to tree comparisons [13]. The time complexity of this reduction is $O(n^3)$ (n is the number of nodes in a PDG). CBCD [27] uses four optimizations for comparing two PDGs. Three of them assume that one PDG is small. However, PDGs of methods in a smartphone app could sometimes be very large. The fourth optimization needs extra effort of splitting and merging, which still raises the bar substantially for handling billions of opcodes. AnDarwin [10] uses locality sensitive hashing (LSH) to find similar connected components of PDGs. However, when a single operation is added or removed, the LSH will change (cannot keep it closer). Its false positive rate is 3.72% for full app clone detection. For core functionality clone detection, its false positive rate could be very high.

Hash-based approaches are efficient to detect code fragments clones [53, 17]. However, they generate many false negatives at handling Type 2 and Type 3 clones.

Kim *et al.* proposed a symbolic-based approach [23] to capture semantically equivalent procedures. But simplifying and comparing symbolic values are not scalable enough.

Motivation: In sum, string-based, token-based, AST-based and hash-based approaches are not accurate for detecting app clones. PDG-based and symbolic-based approaches are accurate, but not scalable enough for app clone detection.

Table 1: Comparison of Literature Work

	Scalability†	Accuracy
String ([5])	$O(L)$	Not accurate for Type 2, 3 clones
Token ([22, 28])	$O(L)$	Not accurate for Type 3 clones
AST ([7, 6, 20])	$O(L)$	Not accurate for Type 3 clones
PDG ([31, 13, 9])	$O(n^2 \cdot M^2)$	Robust for Type 1, 2, 3 clones
PDG-Hash ([10])	$O(NlogN)$	Can handle Type 1, 2, 3 clones, FPR=3.72% for full app clones. For core functionality clones, the FPR could be very high.
Hash ([53, 17])	$O(M^2)$	Not accurate for Type 1, 2, 3 clones
Symbolic-based ([23])	$O(s^2 \cdot M^2)$	Can handle Type 1, 2, 3 clones, FPR > 10%
Centroid	$O(c \cdot M)$	Accurate for Type 1, 2, 3 clones

†: Different implementations may have different time complexity. We show the minimum one. L: the number of statements in all apps. M: the number of methods in all apps. N: the number of apps. n: the number of PDG nodes. s: the number of memory states. c: a small number ($c << M$. Based on our evaluation, $c \approx 7.9$). FPR: False Positive Rate.

Problem statements: *How to achieve accuracy and scalability simultaneously in detecting app clones on Android markets?*

1.3 Insights and Our Approach

Our approach is based on two insights.

Insight 1. If a graph-based approach can avoid graph isomorphism analysis, then it could become both scalable (at code fragment level) and accurate in handling Type 2 and Type 3 clones. We found that a dependency graph could be encoded in a particular way to avoid graph isomorphism analysis while achieving the same clone detection goal.

Insight 2. To handle the billion opcode problem, code fragments level scalability is necessary, but not enough. Scalable cross-market clone analysis requires scalable pairwise comparison between all the methods (i.e., code fragments in multiple markets). The complexity of this comparison is as high as C_M^2 (M is the total number of methods). We found that ideal complexity reduction is possible. Through such reduction, we could reduce the complexity from $O(C_M^2)$ to the level of $O(c \cdot M)$ ($c << M$).

Inspired by the insights, we develop a three-step approach to detect app clones as follows.

Step 1: Geometry-characteristic-based encoding of dependency graph. We develop a special encoding approach which can accurately compare graphs without using isomorphism analysis. Regarding Characteristic 3 of app clones, if part of a PDG is sufficient to capture the Type-2 and Type-3 clones[2], it is not necessary to encode the whole PDG. Based on our observation, we found that we could encode the control flow graph (CFG) part of PDG. We further use a geometry characteristic called *centroid* of CFG to encode a CFG. By comparing centroids instead of the whole CFGs, we can achieve high scalability. In most cases, the information loss caused by encoding may bring inaccuracy. However, we are very surprised to find that centroids, which are encoded using both CFG structures and opcodes, are still very accurate. We refer to the surprising finding as the "*centroid effect*". This observed property of centroids make CFG comparison both scalable and accurate.

Step 2: Localized global comparison. We found that centroids have another special property denoted "monotonic-

[1]Roy et al. [39] define four types of clones. Type 1: Identical code fragments except for variations in whitespace, layout and comments. Type 2: Syntactically identical code fragments except for variations in identifiers, literals, types in addition to Type 1's variations. Type 3: Copied code fragments with further modifications such as changed, added or removed statements in addition to Type 2's variations. Type 4: Code fragments that perform the similar computation but implemented through different syntactic variants.

[2]Capturing Type-1 clones is obvious.

ity". When a method changes a little, its centroid will not change a lot. That is, a large difference between two centroids shows that the corresponding methods are very unlikely to be clones. "Monotonicity" can localize the cross-market method comparison to a small number of methods, which reduces the time complexity of pairwise comparison from $O(C_M^2)$ to $O(c \cdot M)$.

Step 3: Core-functionality-based grouping. Different from most of other works, we use an asymmetrical mechanism to group app clones. For two apps, as long as a large portion of the core functionalities of App 1 are cloned to App 2, we assume that they are app clones no matter how many core functionalities in App 2 are in App 1. This mechanism could detect app clones that add lots of ads and libraries.

1.4 Main Contributions

In summary, we make the following contributions.

- To the best of our knowledge, this is the first work that takes a centroid-based approach to clone detection.
- *Accuracy: We observe the "centroid effect".* It has a surprising "capability" to distinguish cloned methods.
- *Scalability: We find the "monotonicity" property of centroid.* It enables us to achieve $O(c \cdot M)$ time complexity for pairwise comparison.
- We designed and implemented the app clone detection system.
- We evaluated our system on five Android markets. The results demonstrate that our approach achieves high accuracy and high scalability simultaneously.

2. OVERVIEW

We propose a new app clone detection approach that is both accurate and scalable. Using this approach, we can detect potentially app clones across different Android markets in very short time.

2.1 A Motivating Example

We looked into the bytecode in app clones. Due to the page limit, we only show one example. Figure 1 shows two corresponding methods in two app clones. The apps have different names and are deployed in different markets. The only difference between the two methods is that one method in Figure 1(a) adds a function call (marked with an arrow) to initialize an object "touydig". We viewed the code inside "touydig" and found that it will start several ads.

2.2 Architecture

Motivated by the example in Figure 1, at a high level, we use a *bottom-up approach* to detect app clones across different Android markets. We compare different apps after performing 1-to-1 method-level comparisons in multiple markets. Figure 2 shows the architecture of our approach.

Firstly, after preparation (including downloading all the apps from multiple markets and extracting methods from the apps), we encode a projection form of CFG (3D-CFG, Subsection 3.1) to get the "centroid" geometry characteristic of a CFG. This centroid could be viewed as the "mass" center of a 3D-CFG.

Secondly, we measure method-to-method similarity using centroids in multiple markets (Subsection 3.2). Leveraging

```
.class public Lcom/ladty/gjump/GStart;.
method public onCreate(Landroid/os/Bundle;)V
   invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
   invoke-static {p0}, Lcom/gamegod/touydig;->init(Landroid/content/Context;)V
   new-instance v0, Ljava/lang/Thread;
   ......
.end method
```

(a) com.gamelin.gjump_17118600_0.apk (from the "Anzhi" market)

```
.class public Lcom/ladty/gjump/GStart;
.method public onCreate(Landroid/os/Bundle;)V
   invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

   new-instance v0, Ljava/lang/Thread;
   ......
.end method
```

(b) jxty_1367648062215.apk (from the "Dangle" market)

Figure 1: Two corresponding methods in two app clones are from different markets. The first method has one more function call to initialize several ads.

the monotonicity of centroid-based comparison, we could reduce the complexity from $O(C_M^2)$ to $O(c \cdot M)$ (M is the number of methods in multiple markets and $c << M$).

Thirdly, we detect app clones by core-functionality-based grouping based on the method-level comparison results (Subsection 3.3). Similar apps from the same authors/companies or developed using the same frameworks/common third-party libraries may not be real clones. We use *Clone Groups (C-Groups)* to separate the real app clones from similar but not cloned apps.

2.3 Two Properties of Centroid

The observed *centroid effect* and the inherent *monotonicity* property make our approach unique. The centroid effect makes method-to-method comparison accurate and scalable in terms of method size. The monotonicity property makes pairwise comparison in multiple markets scalable in terms of the total number of methods.

Centroid effect. For any two methods in the Android market, they can form a *method pair (mp)*. A *mp* is either *cloned* or *not-cloned*. The goal of method-level comparison is to separate cloned pairs from not-cloned pairs. In other words, the goal is to accurately "pick" cloned pairs out of the whole set of *mp*s. We find that centroids have an amazing "capability" in doing this "picking" job. On one hand, if two methods in a *mp* have the same centroid, this *mp* is almost certain to be cloned. On the other hand, if two methods in a *mp* have different centroids, the *mp* is 99% to be not-cloned based on our experiments. This means maximally 1% of not-cloned *mp*s are missed. We refer to it as the "*centroid effect*". This surprising and intriguing "centroid effect" enables us to achieve high accuracy without sacrificing scalability when detecting cloned methods.

Monotonicity. Besides the "centroid effect", we find that centroid has another property "monotonicity" when solving this problem, which gives us high scalability. When a method changes a little, its centroid will not change a lot. This property of centroid makes the similarity between two methods monotonically correlate to the difference between two centroids. Therefore, after we sort the methods in multiple markets by their centroid values, we only need to compare one method with its "equal class" in the sorted list. We do not need to perform global comparison. Thus, monotonicity enables us to decrease the time complexity of pairwise comparison without sacrificing accuracy.

2.4 Can CFGs Detect App Clones?

PDG can effectively detect Type 2, Type 3 and Type 4 clones. However, in Android markets, Type 4 clones rarely

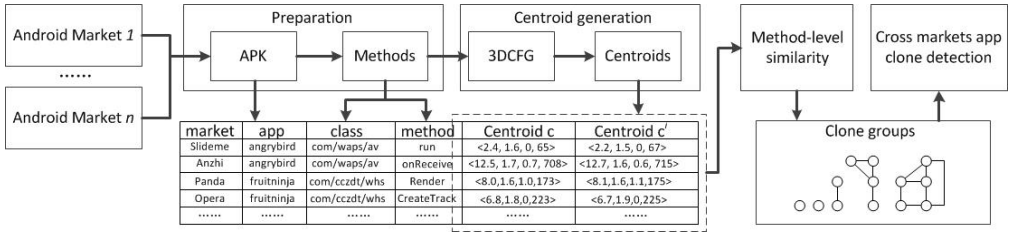

Figure 2: An architecture of our approach. After extracting methods from Android apps (APK files), we generate centroids for those methods. Then we perform app clone detection on the basis of centroids.

exist in app clones. The main reason is that attackers usually do not bother to understand and perform advanced transformations on the bytecode of legitimate apps. Type 2 clones do not impact CFGs. Type 3 clones will still keep the main structures of CFGs. Thus, it is possible to use CFGs to detect clones on Android markets. Compared to PDGs, a clear advantage of CFGs is that CFG-based method-level similarity checking is by nature much simpler and more efficient. CFGs contain much less information than PDGs.

When CFGs are used, one could be concerned with the possibility of resulting in a high false positive rate. Different methods may have the same CFG. Fortunately and surprisingly, we find that the false positive rate could be brought near to zero by combining the CFG structures and the information of opcodes.

2.5 Key Results

We fully implemented a prototype and systemically evaluated it on five real Android markets.

Accuracy: Our approach is very accurate at both method-level and app-level.

Scalability: 1) For five Android markets (including 150,145 apps, 203 million methods and 26 billion opcodes), it takes less than one hour to perform cross-market app clone detection after generating centroids only once. 2) For a given method, it takes less than 0.1 second to find the method clones from the 203 million methods.

3. OUR APPROACH

We use a bottom-up approach to detect app clones. Based on the results of 1-to-1 method-level comparisons, we compare different apps and perform cross-market analysis.

3.1 Centroid of CFGs

We use a vector, called *centroid*, to digest a CFG. It is a geometry characteristic of a CFG.

3.1.1 Physical Model and Challenge

CFG is the control flow graph of a method. Each node in a CFG corresponds to a basic block in the method. A basic block is a straight-line piece of code with one entry point and one exit point. Jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. The code could be source code like JAVA, assembling code like SMALI, machine code like arm instructions or bytecode like DEX (Dalvik Executable).

Remark. (Physical model.) To digest and compare CFGs with both high accuracy and high efficiency, we borrow the ideas of centroid from physics. In physics, especially when analyzing forces on a physical object, people usually use the center of mass, or the *centroid*, to represent an object. It is simple for force and motion analysis since people do not need to consider the structure of the object. When two objects are identical, their centroids are also the same. When the structure of an object changes only a little, its centroid will not vary a lot. Similarity, a CFG may also be viewed as an object. Nodes in the CFG could be viewed as balls. Edges in the CFG could be viewed as sticks. So the whole CFG is very like an object with several balls connected by sticks. Since a centroid can be used to stand for the object, it may be possible to use the centroid to stand for the CFG.

Challenge. The concept of centroid cannot be perfectly transferred from physics to method comparison. In physics, the structure of an object does not change. So its centroid is deterministic. But CFGs never count how long the edge would be. Thus, strictly speaking, a CFG is an object with several variable-position balls connected by variable-length sticks. So its centroid is also variable, which makes it not suitable for comparison.

Thus, we cannot directly extract the centroid from a CFG. We need an intermediate form of representation. That is, we need to project CFGs onto a 3-dimensional space. Then the nodes would get coordinates, and the lengths of edges would be determined automatically. This new type of CFG is referred to as 3D-CFG.

3.1.2 3D-CFG

Android apps are written in Java, which is a kind of structured programming. Sequence, selection (or branch) and repetition are three basic structures of structured programming [49]. We use these three basic structures to encode/transform a CFG to a 3D-CFG.

Definition 3.1 (3D-CFG) *A 3D-CFG is a CFG in which each node has a unique coordinate. The coordinate is a vector $< x, y, z >$. x is the sequence number in the CFG. y is the number of outgoing edges of the node. z is the depth of loop of the node.*

This *sequence number* should make sure that the same node in a CFG will always get the same x-coordinate. Then we could get a 1-to-1 mapping between a CFG and a 3D-CFG. We give each node a sequence number according to the order in which it executes. The first node starts with number 1. If a branch node has sequence number n, we give $n + 1$ to the first node in the branch with more nodes. If two branches have the same number of nodes, we give $n + 1$ to the first node in the branch with more statements. If the numbers of statements are also the same, we give $n + 1$ to the first node in the branch whose first statement has larger binary value. We continue to give sequence numbers to the nodes in one branch until meeting the immediate post-dominator of the branch. Then we go to the other branch and continue to allocate sequence numbers until the last

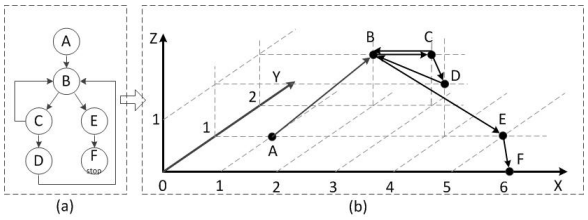

Figure 3: A 3D-CFG example. Figure (a) is a CFG. We add a *stop node* to the end of CFG (node F). Each node in the CFG corresponds to a basic block in the method. Figure (b) shows the 3D-CFG.

node of the CFG. We manually add a *stop node* to the end of the CFG. "Return" statements will flow to the *stop node*. In this way, every method has only one exit.

Example. Figure 3(a) shows the CFG and the 3D-CFG of a real method. Node A in the CFG is the starting node. Its sequence number is 1. It has only one outgoing edge and it is not in any loop. So its coordinate is $< 1, 1, 0 >$. Node B is the second node to execute in this method. It has two outgoing edges and it is also in a loop. So its coordinate is $< 2, 2, 1 >$. B is a branch node. Its immediate post-dominator is node E. The branch starting with C has two nodes. The other branch does not have any node. So the sequence number of node C is 3. We add a stop node in the end. Its coordinate is $< 6, 0, 0 >$.

3.1.3 Centroid Definition

Now we are ready to define the *centroid* of a 3D-CFG. A 3D-CFG can be viewed as a set of nodes connected by edges. It is similar to an object with several balls connected by sticks in physics. Let's assume the weight of the ball is not zero and the weight of the stick is zero. We define the centroid of a 3D-CFG.

Definition 3.2 (Centroid) *A centroid $\vec{c}$ of 3D-CFG is a vector $< c_x, c_y, c_z, \varpi >$. $c_x = \frac{\sum_{e(p,q)\in 3D\text{-}CFG}(\varpi_p x_p + \varpi_q x_q)}{\varpi}$, $c_y = \frac{\sum_{e(p,q)\in 3D\text{-}CFG}(\varpi_p y_p + \varpi_q y_q)}{\varpi}$, $c_z = \frac{\sum_{e(p,q)\in 3D\text{-}CFG}(\varpi_p z_p + \varpi_q z_q)}{\varpi}$ and $\varpi = \sum_{e(p,q)\in 3D\text{-}CFG}(\varpi_p + \varpi_q)$.*

In the equation, $e(p, q)$ is an edge in the 3D-CFG. This edge connects two nodes p and q. $< x_p, y_p, z_p >$ is the coordinate of node p. ϖ_p is the number of statements in the basic block of p. It corresponds to the weight of a ball of a physical object. After computing, the centroid of the 3D-CFG could be viewed as the "mass" center of the 3D-CFG. We use the 3D-CFG in Figure 3(b) as an example. Suppose the values of ϖ_A to ϖ_F are 5, 3, 4, 2, 1 and 0. $\varpi = (5+3)+(3+4)+(3+1)+(4+3)+(4+2)+(2+3)+(1+0) = 38$. The value of x in centroid is $c_x = \frac{5\times1+3\times2\times5+4\times3\times3+2\times4\times2+1\times5\times2}{38} = 2.5526$. In this equation, 5 in 5×1 is the number of statements in node A. 1 in 5×1 is the x part of the coordinate of node A. 3 and 2 in $3\times2\times5$ are similar. 5 in $3\times2\times5$ means that node B appears 5 times in all edges in the 3D-CFG. Similarly, we can get $\vec{c} =< 2.5526, 1.7105, 0.8158, 38 >$. By increasing the number of digits after the decimal point, the precision can be increased.

The definition of the centroid could be extended. ϖ is defined as the number of statements in a basic block. We could define "$\varpi^i = \varpi +$ `# of 'invoke' in the basic block`". So ϖ^i is impacted by the `invoke` statement in the method. We give higher weights to `invoke` statement since Android apps are highly dependent on the underlying framework APIs. We can use ϖ^i to get the centroid $\vec{c}^i$. By combining $\vec{c}$ and $\vec{c}^i$, we could decrease the false positive rate. We can use other types of statements to further extend the definition of the centroid. In this paper, we only use $\vec{c}$ and $\vec{c}^i$ because they are accurate enough to distinguish millions of methods. We use a centroid vector $< \vec{c}, \vec{c}^i >$ to represent a method.

3.1.4 Monotonicity of Centroids

We leverage the *monotonicity* property of centroids to achieve high accuracy and high scalability. This property includes two parts: P1) Two same methods have the same centroid. P2) When a method changes a little, its centroid will not change a lot.

P1 keeps the correctness of the centroid approach. If two of the same methods have different centroids, a high false negative rate may result. P2 makes the similarity between methods m_1 and m_2 monotonically correlate to $|\vec{c_1} - \vec{c_2}|$. When $|\vec{c_1} - \vec{c_2}|$ is smaller, the similarity between two methods will be more. When $|\vec{c_1} - \vec{c_2}|$ is larger, there will be less similarity.

Remark 1. Most approaches (e.g., hash-based and symbolic-based) do not meet P2. When only one statement is changed in a method, we cannot estimate how much a hash sequence or a symbolic value will change.

Remark 2. Centroids are sortable. Based on P1 and P2, after sorting the centroids, we only need to compare a centroid with its neighbors, but not all the centroids. That is, we are able to localize the global pairwise comparison to a small number of centroids, which dramatically decreases the time complexity of pairwise comparison.

3.2 Method-Level Similarity

Centroid Difference Degree (CDD) for two centroids c and c' is defined as follows: $CDD(\vec{c}, \vec{c}') = max(\frac{|c_x - c'_x|}{c_x + c'_x}, \frac{|c_y - c'_y|}{c_y + c'_y}, \frac{|c_z - c'_z|}{c_z + c'_z}, \frac{|\varpi - \varpi'|}{\varpi + \varpi'})$.

CCD is the normalized distance for each dimension. It can directly measure the difference between two centroids. Suppose there are two methods $m{<}\vec{c}, \vec{c}^i{>}$ and $m'{<}\vec{c}', \vec{c}^{i\prime}{>}$. *Methods Difference Degree* $MDD = max(CDD(\vec{c}, \vec{c}'), CDD(\vec{c}^i, \vec{c}^{i\prime}))$.

MDD can be used to compare methods. If two methods are the same, $MDD = 0$. This meets P1 of monotonicity. If a statement is changed inside a basic block, the centroid $\vec{c}$ will not change. If we use the extended centroid such as $\vec{c}^i$, we may find the difference. Changing the sequence of two statements inside a basic block does not change the centroid of the method. From the definition of the centroid, adding or removing a statement will make little impact to the centroid. If more changes are made, $\vec{c}$ will also change more. Thus, the centroid approach meets P2 of monotonicity, which makes it suitable for method comparison.

3.3 Using Centroid to Detect App Clones

Challenge: Method-level similarity could be used to get similar apps. However, similar apps are not always app clones. For similar apps, they usually fall into the following three categories. C1) Apps from the same author; C2) Apps developed using the same framework or using common third-party libraries (e.g. advertisement libraries); C3) App clones. We need to separate C3 from C1 and C2.

To address this challenge, (1) we first use *Application Similarity Degree* to measure how similar two apps are; (2) us-

ing an *ASD* threshold and *Clone Groups (C-Groups)*, we separate category C1, C2, C3 apps from other apps in the market; (3) we remove C1 by comparing public keys of apps; (4) we further remove C2 using a library whitelist. Then what remains is C3.

(1) *Measure app-level similarity.*

App-level similarity is measured using the results of method-level similarity.

Definition 3.3 (Application Similarity Degree) *For two apps a_1 and a_2, Application Similarity Degree (ASD) measures the extent to which a_2 contributes its methods to a_1. $ASD(a_1, a_2) = |a_{2,1}|/|a_2|$. $|a_1|$ is the number of methods in a_1. $a_{2,1}$ is a set of methods. For each method m_i in the set, it meets the following two conditions: 1) $m_i \in a_2$; 2) There is at least one $m' \in a_1$ that satisfies $MDD(m', m) \leq \delta$. δ is a threshold for method-level similarity. $|a_{2,1}|$ is the number of methods in it.*

Note that $ASD(a_1, a_2)$ may not be equal to $ASD(a_2, a_1)$. We do not use a symmetrical definition. This is because *ASD* will not change when an app clone adds lots of ads or libraries. It is very common for an app. For example, we find an app named `com.fgdfhghgh.turbofly_10255200_0` has at least 22 ads. Current definition of *ASD* can detect this situation. But a symmetrical definition of similarity between two apps cannot detect this. Using *ASD*, one can know the percentage of methods in a_2 that are cloned to a_1.

(2) *Get C1, C2 and C3.*

A threshold Δ can separate C1, C2 and C3 from the other apps in multiple markets due to this insight: for any two apps, if they do not belong to C1/C2/C3, then their *ASD* will not be high.

(3) *Remove C1 and C2.*

Three observations of C1 and C2 are found. 1) For an app in Android markets, it uses the author's private key to sign each file and stores the corresponding public key inside the app for verification. If apps are developed by different authors, their public keys are different. 2) Apps developed using the same frameworks (e.g. PhoneGap) or using common third-party libraries directly store the libraries inside the app. 3) When two apps a_1 and a_2 have very different sizes (number of opcodes), $ASD(a_1, a_2)$ could be very high if the big app uses a large portion of the methods in the small app. However, this does not indicate app clones because the big app usually supplies significant more functionalities than the small one. Based on the three observations, we use the concept of *C-Groups* to separate C3 from C1 and C2.

Definition 3.4 (Clone Group) *A Clone Group (C-Group) is a set of apps. It meets the following four conditions.*

- *For any two apps a_1 and a_2, if $MAX(ASD(a_1, a_2), ASD(a_2, a_1)) \geq \Delta$, they are in the same C-Group. Δ is a threshold for ASD.*
- *For any app in Android markets, it is at most in one C-Group.*
- *No two apps in the C-Group have the same public key. We do not check which app is the original one.*
- *Due to the observation (3), the size of small app is at least two-thirds of the bigger app.*

From condition 1 in Definition 3.4, for any app a_1 in a *C-Group*, there is at least one app a_2 in the *C-Group* such that $MAX(ASD(a_1, a_2), ASD(a_2, a_1)) \geq \Delta$. From condition 2, two C-Group can never share one apps. From condition 3, any two apps in a C-Group have different public keys. Based on observation (1), no two apps in the C-Group are from the same author. So we can remove C1. We do not use the app names or the author information in the market website since they can be spoofed.

ASD does not compare the methods in the library whitelist. Thus, *ASD* will not be impacted by those libraries. So C2 could be removed by a suitable Δ. We manually identify some known libraries (e.g., "`android/support`). The full list is in [8]. If libraries outside the list make two apps be falsely viewed as app clones, it is a false positive. We evaluate the false positives in Subsection 5.2.3.

4. IMPLEMENTATION

We implement a prototype to do preparation, to generate centroids and to perform cross-market app clone detection. This prototype includes about 7,000 lines of C++ code and 500 lines of python code.

In the preparation step, we use a python script to download apps (APK files) from five Android markets. Then we transform APK files to SMALI code using the tool called `baksmali` [43]. It is a disassembler for Android's DEX format. We use this SMALI code since it could be assembled to runnable DEX format again. This code type is an attractive intermediate form for app clone. We parse the SMALI code and generate 3D-CFGs.

In the centroid generation step, we give coordinates to the nodes in 3D-CFGs and compute the centroids. Then we put the centroids into a database. Each record in the database corresponds to a method. It contains the market name, app file name, class name, method name and centroids (Figure 2). We generate SMALI code and centroids of different apps in parallel. Thus, when more computers are used, the analysis time will be decreased. We use an incremental approach to update the database. When a new app is added into the market, we only need to generate centroids for methods in the app and insert them into the database.

In the clone detection step, we compare each method in one app with all the methods in other apps. If one uses SQL queries to do the comparisons, we find that is very slow. So we use in-memory comparisons. Considering the two properties of centroids, one can compare centroids after sorting them. So the time complexity for comparison will decrease from $O(M^2)$ to $O(c \cdot M)$. M is the number of methods in multiple markets. $O(M^2)$ is the time complexity of traditional pairwise comparison. c is the average number of methods with which one method needs to compare. Usually, $c << M$. Based on our evaluation, $c \approx 7.9$.

5. EVALUATION

We evaluated our approach on five typical third-party Android markets: two American markets (Pandaapp [37] and Slideme [42]), two Chinese markets (Anzhi [3] and Dangle [11]) and one European market (Opera [36]). We performed a cross-market analysis to find app clones. Our experiments were conducted on Lenovo Thinkcenters with CORE i5 3.2GHz CPU and 4GB of memory.

5.1 Cross-Market App Clone Detection

The number of apps in C-Groups and the number of C-Groups give a direct impression of the app clones in An-

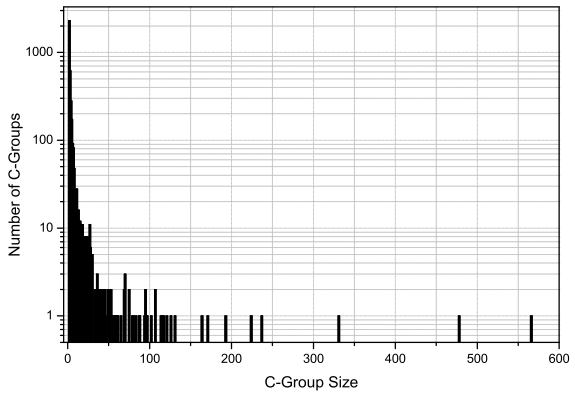

Figure 4: Histogram of the C-Group sizes on logarithmic scale.

droid markets. Two C-Groups never share the same app. Δ, the threshold for application similarity degree, impacts the results. We use $\Delta = 0.88$ to perform the cross-market app clone detection. When $\Delta = 0.88$, the measured false positive rate through 100 manual examination of randomly selected C-Groups is 0% (Subsection 5.2.3).

How many apps are in C-Groups? When an app is in a C-Group, it means that either this app is an app clone or at least one other app clones this app. For the five markets, we detected 3,916 C-Groups and in total 20,292 apps are in these C-Groups. That is, 13.51% of the apps in all five markets are in C-Groups.

Among the 20,292 apps, 1,416 apps are from the Pandaapp market, which takes 13.54% of the apps in Pandaapp. 2,022 apps are from the Slideme market (14.23% of Slideme), 9,850 apps are from the Anzhi market (18.39% of Anzhi), 4,268 apps are from the Dangle market (19.59% of Dangle), and 2,736 apps are from the Opera market (5.46% of Opera). The ratio for the two Chinese Android markets (Anzhi and Dangle) are higher than others. The ratio for the two American Android markets (Pandaapp and Slideme) are on average. The European market has the lowest ratio.

Histogram of the C-Groups. The number of apps in C-Groups gives an overview of the app clones. We use the histogram of C-Groups in Figure 4 to show the details. The x-axis shows C-Group size (the number of apps in a C-Group). The y-axis shows the number of the C-Groups with a certain size. From the figure, we find that the majority of C-Groups consist of small number of apps. 59% of C-Groups contain two apps. Over 90% of C-Groups contain less than 7 apps. Only 0.4% of C-Groups contain more than 100 apps. The largest C-Group consists of 566 apps.

How many C-Groups are cross-market? If a C-Group contains apps from different markets, we say the C-Group is cross-market. After analyzing the C-Groups, we find 1,779 out of 3,916 C-Groups are cross-market. That is, 46% of all the C-Groups are cross-market C-Groups. This shows that attackers would like to publish the app clones in different markets for more benefits. This also justifies the necessity to do cross-market app clone detection. Figure 5 shows the cross-market app clone situation. The five nodes are corresponding to the five markets. The number beside an edge shows how many C-Groups cross the two markets. We find that the two Chinese markets have more cross-market C-Groups than other markets.

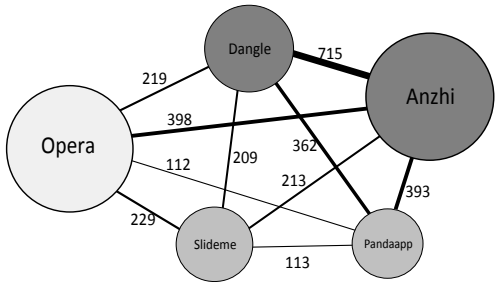

Figure 5: Cross-market app clone situation. The five nodes are corresponding to the five markets. The size of a node is proportional to the number of apps in a market. Two markets with deep color (Dangle and Anzhi) are Chinese markets. The market with light color (Opera) is from Europe. The other two markets are American markets. The number beside an edge shows how many C-Groups cross the two markets. The thickness of an edge is proportional to the number.

5.2 Accuracy

To get good accuracy, application similarity degree (ASD, in Subsection 3.3) and its threshold Δ must work well. In other words, they need to meet the following two requirements: R1) ASD needs to measure the unbiased similarity; R2) *C-Groups* should separate C3 from C1 and C2. To satisfy R1, method-level accuracy is essential. So we measure the false positive rate and false negative rate at method-level in Subsection 5.2.1 and 5.2.2. To satisfy R2, we manually verify the detected app clones (false positive) in Subsection 5.2.3.

5.2.1 False Positive Rate at Method Level

The centroid approach gives out similar method pairs. We need to check whether they are real clones. If not, false positives occur. Checking not-cloned pairs needs manual effort. However, even for a small testset (10^3 apps with 10^6 methods), the number of similar method pairs usually exceeds 10^8. If we randomly select a subset (e.g., 1000 pairs) and manually check them, it is not representative. Neither can we manually check all the pairs. So an automatic mechanism to approximate the number of real clones is necessary.

We use the *longest common subsequence (LCS)* [18] of opcodes for approximating the number of method clones. LCS is widely used in clone detection (e.g., [53]) and plagiarism detection (e.g., [51]). Thus, we use LCS to compute the similarity between methods. For two methods m_1 and m_2, we view each method as a sequence of opcodes. Suppose l_1 and l_2 are the numbers of opcodes in m_1 and m_2, respectively. If the LCS between m_1 and m_2 contains l_{max} opcodes, the similarity between m_1 and m_2 is $\frac{l_{max}}{max(l_1,l_2)}$. We use the same threshold (70%) used in [53].

We select 500 apps randomly from each Android market to form a testset ("Andr2500"). Then we use these 2,500 apps to estimate the method-level false positive rate ($MFPR$). The size of the testset is 13.2GB. It contains 2,633,144 methods and 478,096,757 opcodes. The curve in Figure 6 shows the estimated $MFPR$. When the threshold δ (Subsection 3.3) for method-level similarity is less than 0.01, $MFPR < 1\%$. When $\delta = 0$, $MFPR = 0.38\%$.

5.2.2 False Negative at Method Level

To get method-level false negative rate ($MFNR$), we need to count how many method clones are not identified by our approach. However, we could not find such a benchmark

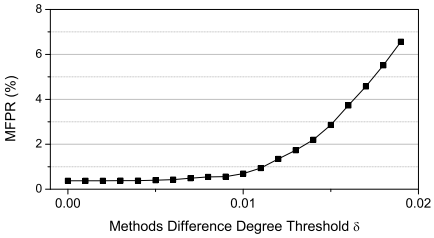

Figure 6: The estimated *MFPR*. When $\delta = 0$, *MFPR* = 0.38%.

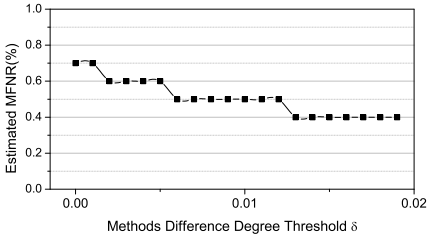

Figure 7: MFNR decreases with the increas of δ. All the false natives are small CFGs (with less than 4 nodes).

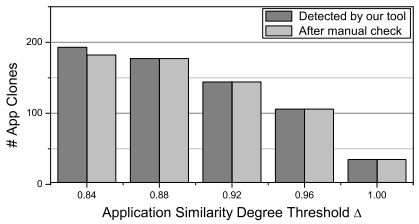

Figure 8: We manually check the app clones ("Andr2500" testset) reported by our tool.

for Android apps. Moreover, generating method clones by arbitrarily transforming methods may not capture the characteristics of real clones on Android market. Neither can we check all the method pairs in "Andr2500" testset (over 10^{12}) even using the automatic mechanism. So how to use an unbiased way to get the ground truth of method clones which could also represents real clones on Android markets?

To get the unbiased method clones, we first get the ground truth of app clones by manually installing and comparing apps. Note that the results of app clones are not based on any method level analysis, which will make our test more unbiased. Secondly, we manually compare the methods of these app clones and get the method clone pairs. At last, we use the centroid-based approach to detect method clones. Suppose the number of method clone pairs manually verified by us is mp and the number of detected pairs is mp_d. Then $MFNR = 1 - mp_d/mp$.

To increase the possibility to find app clones, we download apps from two aspects: A1) similar names, or A2) similar descriptions or keywords such as "sudoku". Attackers would also like to use A1 and A2 to let users easily find the app clones. After downloading, installing and comparing the apps, we got 112 app clones (in 40 C-Groups) with 62,121 methods. We randomly select 1,000 methods and manually find the method clones in the corresponding app clones. After manually identifying method clones, we randomly select 1,000 pairs, which serves as the unbiased ground truth.

Figure 7 shows the results. From the figure, we find that $MFNR$ decreases with the increase of δ. When $\delta = 0$, the $MFNR$ is 0.7%. When δ increases to 0.02, the $MFNR$ is 0.4%. We checked these method pairs and found they are all small CFGs (with less than 4 nodes).

Why the centroid approach has very low *MFNR* with controllable *MFPR*? It has a unique "capability" to distinguish cloned methods. Based on the false negative test results (Figure 7), we find that the method pairs with $\delta = 0$ take more than 99% of all the clone pairs. When a method pair is detected as method clones, the probability that these two methods have the same centroid is 99% or higher. This is the reason why both $MFNR$ and $MFPR$ are so low. When δ is non-zero, the number of cloned method pairs is surprisingly low (less than 1%). This means when the centroids of two methods m_1 and m_2 are different, the probability that m_1 and m_2 are a cloned pair is less than 1%. So the centroid approach has the capability to distinguish method clones. One main reason is that centroids keep part of structure information of CFGs. We call it as the "*centroid effect*". It is a unique characteristic to make both the $MFNR$ and $MFPR$ low.

5.2.3 Accuracy in Detecting App Clones

We first use "Andr2500" as the training testset to get the suitable value of threshold Δ for application similarity degree. Then we use this value to detect app clones in all the five markets and evaluate the false positive rate.

(1) Get Δ from a Training TestSet

The value of Δ impacts how well our approach separates app clones from other apps. Recall that we use C-Groups to perform this separation. Given the C-Grouping results, if an app in a C-Group is not an app clone, we call it a false positive. So we can use the number of false positives to choose suitable Δ.

We use "Andr2500" as the training testset and get C-Groups by different Δ. For each app in the C-Groups, we manually install it and look into the SMALI code. Then we compare it with other apps in the same C-Group. If it is not really an app clone, a false positive occurs. Figure 8 shows the results. For each Δ, the left bar shows the number of app clone detected by our tool and the right bar shows the number of app clones after manual check. From the figure, the number of false positives decreases with the increase of Δ. But when Δ increases, some manually confirmed app clones can no longer be detected. We choose $\Delta = 0.88$ to detect app clones in all the five markets as the number of false positives and false negatives are both small at this point.

(2) False Positive Rate on Five Android Markets

To get the ground truth of app clones, we manually check the apps in C-Groups. Based on our manual check, we confirmed the apps in the C-Groups. For the five markets, our approach detected 3,916 C-Groups and in total 20,292 apps are in these C-Groups. It is not feasible to manually check all of them. Instead, we randomly select 100 of the 3,916 C-Groups and then manually check each app in the C-Groups. For any app in a C-Group, if it is not really an app clone, we view the whole C-Group as a false positive, which is very conservative. After manually checking the 359 apps in the 100 C-Groups, we did not find any false positive.

Remark 1: In total there are 3,916 C-Groups. If the remaining C-Groups are checked, based on the following observation, the false positive rate should also be around zero. False positives come from two aspects: 1) Wrongly detected method clones. Since apps usually have many methods to support various functionalities, impacting the results of app clones needs a large number of wrongly detected clones. Due to the low false positive rate (0.38%) at method-level of our approach, for a method with 300 methods, only 1 method on average is wrongly viewed as clone. This is almost impossible to impact the result of app clones. 2) Common libraries. If two apps use common libraries and the num-

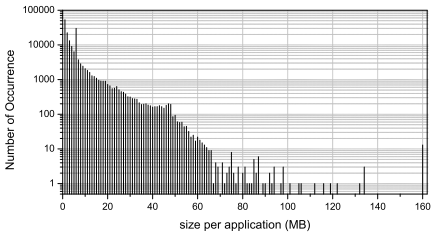

Figure 9: Number of occurrence vs size per app.

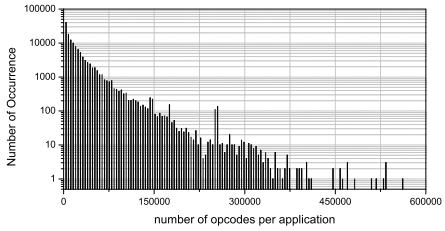

Figure 10: Number of occurrence vs number of opcodes per app

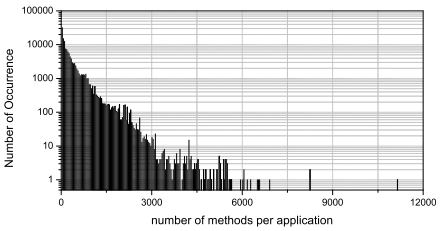

Figure 11: Number of occurrence vs number of methods per app

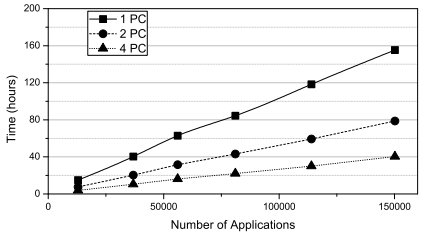

Figure 12: Time to compute centroids. Besides linearly increasing with the number of apps, the time also linearly decreases with the number of computers.

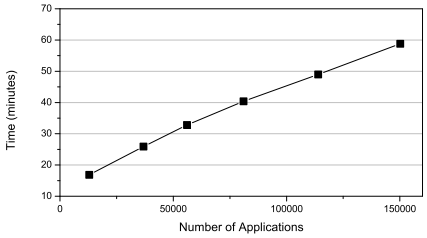

Figure 13: Time to perform app clone detection. This time is almost linear with the number of apps.

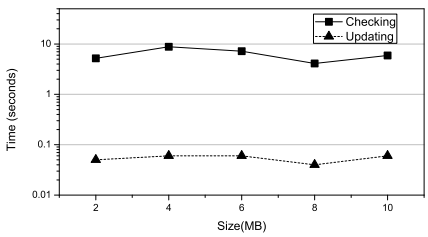

Figure 14: Time to perform clone detection for new apps and time to update the market database.

ber of the methods in the libraries is far more than that of the core functionalities, false positives will occur. This usually happens when an app is very small and it uses a lot of libraries (e.g., ads). However, we whitelist 73 popular libraries, which makes the possibility very low. Moreover, when we add two apps into a C-Group, the size of small app cannot be too small (at least two-thirds of the bigger app). This will further decrease the false positive rate.

Remark 2: We did not attempt to measure the false negative rate since it is not feasible to get the ground truth of app clones (i.e., manually performing 1-to-1 comparison for all the 150,145 apps). However, we compare the results with others [53] and find our approach detects more app clones.

5.3 Scalability

We measure the scalability of our approach from the following three aspects: the scale of the five markets, the performance on cross-market app clone detection and updating.

5.3.1 Dataset Statistics

We analyze all apps in the five markets to gain a general understanding of our dataset. Figure 9 to 11 show the distribution of the sizes of APK files in megabytes, the distribution of opcodes per app and the distribution of methods per app. All these distributions are skewed to the right. About 90% of apps are less than 13MB and 54% of apps are less than 3MB. About 50% of apps have opcodes less than 11,000 and have methods less than 200. The total file size of these APKs is 869GB. The total number of opcodes in all apps is approximately 26 billion. The total number of unique methods in all apps is about 203 million.

5.3.2 Performance on Cross-Market App Clone Detection

Whole market app clone detection can be divided into two steps: centroid generation and clone detection.

Centroid generation. The time needed to generate the centroid depends on the number of apps in the market. This step can be done in parallel. We use 1 computer, 2 computers and 4 computers to measure the time. Figure 12 shows the results. From the figure, we can see that the time linearly increases with the number of apps in the market. The time also linearly decreases with the number of computers. If we use 4 computers, we can generate centroids for the whole market in about 40 hours.

Clone Detection. The performance of clone detection depends on the number of apps. Figure 13 shows the time to detect app clones. From the figure, we find that the time almost linearly increases with the market size. It takes less than an hour to detect app clones on the five whole markets with 150,145 apps.

What is the value of "c" in the time complexity $O(c \cdot M)$? Our approach detects app clones using the results of method-level comparison. Different from other approaches, the property "monotonicity" of centroid localizes the global pairwise comparison to a small number of methods, which dramatically decreases the time complexity of pairwise comparison. The smaller the value of c, the less time is needed for comparison. In this cross-market app clone detection process, we record this value. The average value of c is 7.9. That is, for each method, our approach on average only needs to compare it with less than 8 other methods, instead of 203 million methods. This is the reason why we can process 150,145 apps apps within one hour.

5.3.3 Performance on Clone Detection for New Apps and Market Updating

Adding new apps is very common in Android markets. After a new app is uploaded to the market, it should first be checked whether it is an app clone. In this process, it will be compared with all the apps in the whole market. If it is not cloned, it can be added to the market and become available for downloading. Both the app checking time and database updating time should not be long.

We evaluate the performance using real data. For the centroid database, we use the current database of all the five Android markets. For the newly added apps, we download them from another third-party market (1mobile) [1]. It is a

popular market in the U.S. Considering the size of the new apps may impact the time, we divide them into five groups. Each group has five apps with similar sizes of APK files.

Figure 14 shows the results. The upper curve shows the average app checking time for the group. This process includes the time of generating SMALI files, getting centroids and detecting clone status of the new apps. From the figure, it takes about 8 seconds to finish these steps on average. We also find the time is not linear to the APK file sizes. This may be because resource files take some spaces in the app. Considering there are more than 100 methods in each app on average, it takes less than 0.1 second to find the method clones from 203 million methods. The lower curve shows the database updating time, which is less than 0.1 seconds.

6. LIMITATIONS

Centroid-based approach has several limitations. Firstly, although our approach is extremely effective to detect Type 1, Type 2 and Type 3 clones, it may not be effective to detect Type 4 clones. Type 4 clones require the attackers to understand the code. Type 2 and 3 clones are effective for attackers to achieve their goals. They would probably not pay the effort to understand and perform the advanced transformation on the bytecode of legitimate apps. Actually, we did not find Type 4 clones on Android markets.

Secondly, adding one node in *small* CFGs (with less than 4 nodes) may change the centroids a lot (Attackers may not delete a node from CFGs since they want to keep the original functionalities). But for *big* CFGs (with 4 nodes or above), centroid-based approach is effective. Based on our evaluation, *small* CFGs take about 2.3% of all the opcodes in all the five markets. We believe we can ignore them.

Thirdly, an app clone could evade detection by only cloning a small number of methods in the original app (*partial cloning*). We note that there are no general solutions for handling partial cloning. However, core methods (i.e., functionalities) need to be reused. Otherwise, the cloned app may not work properly. One solution to detect partial cloning is to find the core methods in the apps and only compare these methods. The core methods could be chosen by their sizes such as the number of nodes in the CFGs.

Fourthly, if an app clone has far more opcodes than the original app, we may not detect it (due to the condition 4 in the definition of C-Group). This usually happens when some apps are developed on the basis of open source apps (e.g., sample apps from Android SDK). If they have lots of new functionalities, we should not view them as clones. An exception is that an app is cloned by adding lots of ad libraries. However, the whitelist with 73 popular libraries greatly increases the possibility to detect the app clones. It could also be extended. For the unpopular libraries, attackers would probably not add lots of them at the same time.

7. RELATED WORK

Centroid-based cloning detection is mainly related to three bodies of work as follows.

Clone detection. String-based approaches [5, 4] view each line of source code as a string and detect clones based on matching the sequence of strings. *Token-based* approaches [40, 28, 29, 22] parse a program to a sequence of tokens and compare these tokens to find clones. *AST-based* approaches [24, 47, 50, 7, 6] construct the abstract syntax trees (AST) from two programs and detect clones by finding the common trees. Lee *et al.* [26] introduced a multidimensional token-level indexing using an R* tree on Deckard's vectors [20]. *Graph-based* approaches [31, 12, 25, 13, 27] generate CFGs or PDGs from programs and compare them by subgraph isomorphism. Kim *et al.* proposed a symbolic-based approach [23] to capture semantically equivalent procedures. Detailed analysis of these approaches is in Subsection 1.2.

Detection of similar Android apps. Juxtapp [17] detects code reuse in Android apps by feature hashing. DroidMOSS [53] uses a fuzzy hashing technique to detect app clones. DNADroid [9] detects Android app clones by performing subgraph isomorphism comparison on PDGs. Androguard [2] uses several standard similarity metrics to hash methods and basic blocks for comparison. PiggyApp [52] extracts various features such as the requested permissions of primary modules to detect "piggybacked" apps. AnDarwin [10] splits PDGs to connected components and extracts a vector which contains the number of specific types (e.g., *binary operation* type) for each components. Then it uses locality sensitive hashing to find similar vectors as code clones. Its false positive rate is 3.72% for full app clone detection. For core functionality clone detection, its false positive rate could be very high. CLAN [32] detects related Java apps using API calls, which could be potentially applied to Android apps.

Software birthmark. A software birthmark is a unique characteristic of a program that can be used to determine the program's identity. Birthmark can be divided into two categories: static birthmark and dynamic birthmark. *Static birthmarks* [34, 44] are usually the characteristics in the code that cannot easily be modified such as constant values in field variables, a sequence of method calls, an inheritance structure and used classes. Lim et.al [30] proposed an n-gram flow path birthmark. The bytecodes in n continuous basic blocks are concatenated to construct a possible flow path. To compare two flows, the semi-global alignment algorithm is used. To compare two birthmarks (i.e., two sets of flows), a maximum weight matching is performed on the set of all pairwise comparisons of those flows. The matching sum is used for measuring the similarity. A centroid could be viewed as a novel static birthmark. *Dynamic birthmarks* are usually as follows: whole program path [33] birthmark, sequence of API function calls/frequency of API function calls birthmark [46, 45], call sequences to Java standard API [41], system calls birthmark [48] and invariant value sequences birthmark [19]. They need to dynamically analyze the program, which is not suitable for cross-market app clone detection.

8. CONCLUSION

We present a *centroid-based* approach to detect cross-market app clones in five whole Android markets. The observed "*centroid effect*" and the inherent "*monotonicity*" property enable our approach to achieve both high accuracy and scalability. It takes less than one hour to perform cross-market app clone detection. Regarding future work, more analyses of the detected app clones are yet to be done.

9. ACKNOWLEDGEMENTS

This work was supported by NSF CCF-1320605, ARO W911NF-09-1-0525 (MURI), NSF CNS-1223710, W911NF-13-1-0421 (MURI), NSFC 61100226.

10. REFERENCES

[1] 1mobile. 1mobile android market. http://www.1mobile.com/, 2013.

[2] Androguard. Reverse engineering, malware and goodware analysis of android applications ... and more. http://code.google.com/p/androguard/, 2013.

[3] Anzhi. Anzhi market. http://www.anzhi.com/, 2013.

[4] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

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

[6] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *icsm*, pages 368–377, 1998.

[7] I. D. Baxter, C. Pidgeon, and M. Mehlich. Dms®: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[8] K. Chen. 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.

[9] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. *Computer Security–ESORICS 2012*, pages 37–54, 2012.

[10] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar android applications. In *ESORICS*, 2013.

[11] Dangle. Dangle. http://android.d.cn, 2013.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[13] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330. ACM, 2008.

[14] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *ACM Sigplan Notices*, volume 45, pages 175–190. ACM, 2010.

[15] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *MobiSys*, 2013.

[16] Google. Google play. https://play.google.com/store, 2013.

[17] 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 & Vulnerability Assessment*, 2012.

[18] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

[19] 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. ACM, 2011.

[20] 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*, pages 96–105. IEEE Computer Society, 2007.

[21] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

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

[23] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 301–310. IEEE, 2011.

[24] Y.-C. Kim and J. Choi. A program plagiarism evaluation system. *ICCSA*, pages 221–223, 2005.

[25] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Static Analysis*, pages 40–56, 2001.

[26] 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*, pages 167–176. ACM, 2010.

[27] J. Li and M. D. Ernst. Cbcd: Cloned buggy code detector. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 310–320. IEEE, 2012.

[28] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, volume 6, pages 20–20, 2004.

[29] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, pages 176–192, 2006.

[30] 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. *Information and Software Technology*, 51(9):1338–1350, 2009.

[31] C. Liu, C. Chen, J. Han, and P. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *SIGKDD*, pages 872–881, 2006.

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

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

[34] G. Myles and C. Collberg. K-gram based software birthmarks. In *ACM symposium on Applied computing*, 2005.

[35] D. Octeau, S. Jha, and P. McDaniel. Retargeting

android applications to java bytecode. In *FSE*, page 6. ACM, 2012.

[36] Opera. Opera software. http://apps.opera.com, 2013.

[37] Pandaapp. Pandaapp download center. http://download.pandaapp.com, 2013.

[38] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 447–456. ACM, 2010.

[39] C. Roy and J. Cordy. A survey on software clone detection research. Technical report, Queen's School of Computing TR, 2007.

[40] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[41] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 274–283. ACM, 2007.

[42] Slideme. Slideme: Your marketplace for android apps. http://slideme.org, 2013.

[43] smali. An assembler/disassembler for android's dex format. http://code.google.com/p/smali/, 2013.

[44] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 569–575, 2004.

[45] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. Ichi Matsumoto. Design and evaluation of dynamic software birthmarks based on api calls. *Info. Science Technical Report NAIST-IS-TR2007011, ISSN*, pages 0919–9527, 2007.

[46] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.-i. Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. In *International Symposium on Future Software Technology*, volume 2004, 2004.

[47] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pages 317–325. Australian Computer Society, Inc., 2004.

[48] 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. ACM, 2009.

[49] Wikipedia. Structured programming. http://en.wikipedia.org/wiki/Structured_programming, 2012.

[50] W. Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 2006.

[51] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 111–121. ACM, 2012.

[52] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *CODASPY*, pages 185–196. ACM, 2013.

[53] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *CODASPY*, pages 317–326. ACM, 2012.

[54] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.