Robust App Clone Detection Based on Similarity of UI Structure

YANGYU HU, GUOSHENG XU, (Member, IEEE), BOWEN ZHANG, GUOAI XU, AND MIAO ZHANG

1School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China
2School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Guosheng Xu (guoshengxu@bupt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB0801903, in part by the Foundation Strengthening Program for the Key Basic Research of China under Grant 2017-JCJQ-ZD-043, and in part by the National Natural Science Foundation of China under Grant 61897069.

ABSTRACT App clone is a serious threat to the mobile app ecosystem, which not only damages the benefits of original developers, but also contributes to spreading malware. App clone detection has received extensive attentions from our research community, and a number of approaches were proposed, which mainly rely on code or visual similarity of the apps. However, the tricky plagiarists in the wild may specifically modify the code or the content of User Interface (UI), which will lead to the ineffectiveness of current methods. In this paper, we propose a robust app clone detection method based on the similarity of UI structure. The key idea behind our approach is based on the finding that content features (e.g., background color) are more likely to be modified by plagiarists, while structure features (e.g., overall hierarchy structure, widget hierarchy structure) are relative stable, which could be used to detect different levels of clone attacks. Experiment results on a labeled benchmark of 4,720 similar app pairs show that our approach could achieve an accuracy of 99.6%. Compare with existing approaches, our approach works in practice with high effectiveness. We have implemented a prototype system and applied it to more than 404,650 app pairs, and we found 1,037 app clone pairs, most of them are piggybacking apps that introduced malicious payloads.

INDEX TERMS App clone, code clone, mobile app, user interface, android.

I. INTRODUCTION

Mobile apps have seen widespread adoption in recent years, with over 2.8 million apps in Google Play and billions of downloads [1], [2]. Mobile app market expected to reach $189B by 2020 [3], which attracts millions of developers [4], including malicious developers and hackers.

Software clone exists since PC era and appears on Android platform more extensively owing to the openness and popularity of Android system. In the context of app clone, the plagiarists’ goal is to grab subscribers and gain fame by copying the core function [5], [6], the UI [7]–[9], and even product names and brands of legal apps [10], [11]. Clint et al. [5] reported that repackaging apps will result in a 14% decrease in advertising revenue for the hardworking developers of original apps. At the same time, app clone is a major way of malware distribution in Android platform.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Imran Tariq. Zhou and Jiang [12] reported that 86% of all malware distribution relies on repackaged apps. In addition, the third-party Android markets have proliferated and emerged in large numbers, which are available for the legitimate developers and plagiarists to upload their apps. The review-mechanism of most app markets is inadequate, management based on user feedback information is too late to prevent the spread of app clones. Recent work suggested that app clones and fake apps are recurrently found in Google Play, and it takes a long time for Google Play to remove them [13].

Considering that app clones would introduce great damage to developers and mobile app ecosystem, it is important to identify app clones timely and accurately. Various techniques have been proposed to perform Android app clone detection. Most of them are proposed to solve the problem of app repackaging based on static code analysis [6], [14]–[20], including approaches based on static code fingerprints and graphs (e.g., Program dependency graph) which characterize the code structure. In recent years, techniques that rely
on visual similarity emerged, including approaches based on runtime screenshot [7], [9] and runtime UI birthmarks extracted from widget elements [8], [21].

However, there still remain some limitations in state-of-the-art approaches:

L1 Although static software birthmark based approaches [6], [14]–[17] are faster comparing to dynamic approaches, the practice of code reuse and the increasingly usage of the complex obfuscation technique may greatly affect the detection results [22]–[24]. In addition, an adversary could easily evade the detection without modifying the appearance of the app, such as spoofing attacks by UI clones [9]. Static approaches are inefficient for these new attack vectors.

L2 As the plagiarists would want to keep the look and feel of the UI similar to the original apps, recent researches [7]–[9], [21] mainly focused on detection techniques that rely on visual similarity of apps. These methods can cope with some advanced UI-based attacks, including widgets’ text modification, background picture substitution, and widgets’ size adjustment, etc. However, some plagiarists may contrapuntally modify the widget attributes of the UI to evade detection. As shown in FIGURE 1, plagiarists not only alter the background color of the UI title widget, but add or delete widgets in the bottom part of UI.

To address these limitations, we propose a new approach to detect app clone based on the similarity of UI structure. The UI structure information is used to generate dynamic software birthmark, and can be extracted from view hierarchy information when the activity is in the foreground of the system. The intuition behind our approach is based on the following findings:

F1 Dynamic software birthmarks are accurate and obfuscation-resilient since semantics preserving obfuscation techniques do not affect runtime behaviors.

F2 App clones would likely have UIs that are similar to the original apps so as to leverage on the popularity [7]–[9]. However, content features of UIs are more likely to be modified by plagiarists, with little affecting the runtime appearance of the applications towards the user. While the structure features are relative stable, which could be used to detect different levels of clone attacks.

To summarize, we make the following main contributions in this paper:

• We have proposed a novel approach for android app clone detection based on the similarity of UI structure, which is obfuscation-resilient and efficient even when the content of UI has been changed.
• We have evaluated our approach on several sets of real-world apps by comparing with a state-of-the-art code-based app clone detection approach [25]. Experiment results suggested that our approach performed better, with both low false positive rate and false negative rate.
• We have implemented a prototype system and applied it to more than 404,650 app pairs. Experiments result showed that we could achieve an accuracy of 99.6%, and we have identified various real world app clones.

II. BACKGROUND

A. ANDROID APP

Android apps are generally APK (Android Package) files packaged by developers, which is actually a compressed file in ZIP format. An APK mainly contains Dalvik bytecode, UI code, resource files, configuration files, and signature information. It needs to sign the Android app if the developer wants to publish it, nevertheless developers can use their own certificates to sign the APK files without any authorization, which makes it possible to decompress the app and repackage it.
Compared to applications on other platforms, Android apps have the following specific characteristics:

1. Android app is usually implemented in Java language and compiled into Dalvik bytecode subsequently, and it can also be developed by native code. The plagiarists can implement obfuscation in various ways, such as variable obfuscation, control flow obfuscation, Dalvik bytecode encryption, etc. Thus, it is not feasible to identify these types of clones based solely on static features.

2. Most of Android apps embedded third-party libraries (TPLs) as an auxiliary support [26], such as advertisement library, social networking library, and analytic library. TPLs will occupy a large proportion of the code including function code and UI code, thus will have an impact on accuracy of code-based clone detection [6]. Most of previous works use a whitelist to filter TPLs by comparing the package names [14], [27]. Considering that obfuscation may change the package names, making it harder to filter TPLs. Various approaches have been proposed to address this limitation [6]. Note that such obfuscation techniques do not affect runtime behaviors (e.g., activity name) of apps. Thus, in this paper, we propose to filter TPLs based on the runtime behaviors of activities.

3. Android apps are composed of components with multiple entry points, and the components communicate with each other based on intent. In this paper, we start each activity explicitly using explicit intents, which is demonstrated to be an effective approach by previous work [21]. Even though this method poses a risk of losing information when activities fail to start, it could achieve the balance between UI coverage and the performance.

B. ANDROID GRAPHICAL USER INTERFACE

Android GUI subsystem is composed of various system services and components. Activity is a type of app component that provides one or more windows to users. Each activity that contains the GUI contents to be displayed must be declared in AndroidManifest.xml in APK file. Developers generally define the entry point of each activity in the relevant widgets, and the activity will be triggered by user interaction under certain conditions. In particular, if an activity declares the properties by developer as follows,

\[
\text{category android:name = "android.intent.category.launcher"} \]

it can be started directly through “AM”, a command line tool provided by Android. We utilize this feature to grab the information of UI automatically, and we will illustrate it in detail in Section III.

C. APP CLONES

Android app clone is the case where one app intentionally copy the function, the UI, and even product names and brands of another legal app to grab subscribers and gain fame accordingly. In order to understand the characteristics of app clones more comprehensively and systematically, we classify the clone attacks of Android apps into three types in terms of code and UI, as shown in TABLE 1. We then describe each category in detail.

1. **App Repackage.** Plagiarist decompiles the android app, adds or modifies parts of the code to achieve certain goals. The repackaged app can be generated automatically by sign tool, and released into an app store whose review-mechanism is inadequate. For this kind of clone apps, the functional code and UI code are similar with original ones.

2. **Function Clone.** For the apps whose targeting field and demand are relatively new, plagiarist makes the UI of clones look similar to the legal ones by rewriting the interface code and copying the functional code, so as to mislead users to download. For this kind of clone apps, the functional code are similar with original ones, while the UI code are dissimilar.

3. **User Interface Clone.** Plagiarist rewrites the functional code and the UI code to copy the UI style and UI design logic of legal apps. This attack type generally appears in the same app category, through the excellent UI design to enhance the attractiveness of the users. Hence, the functional code and UI code of this kind of clone apps are dissimilar with original ones.

To evade from detection, the plagiarists are likely to apply automatic code obfuscation techniques to the app clones before signing and publishing it. Besides, an adversary can also contrapuntally modify some UI content features, such as a simple obfuscation tool proposed by [7]. In this paper, our approach is based on the observation that structure features are relative stable, it may require more effort and understanding for the plagiarist to modify the GUI design of original app. We will evaluate the effectiveness of our approach in detecting app clones from these three types of clone attacks, in Section V.

### TABLE 1. Types of android app clone.

| Category      | Functional Code | UI Code | UI Structure |
|---------------|-----------------|---------|--------------|
| App Repackage | Similar         | Similar | Similar      |
| Function Clone| Similar         | Dissimilar | Similar      |
| User Interface Clone | Dissimilar | Dissimilar | Similar      |

III. METHODOLOGY

The overall architecture of our detection approach is shown in FIGURE 2. The detection process can be divided into four stages. Firstly, we modify the declarations of activities in AndroidManifest.xml generated by decompiling the APK file, thus each activity of the modified app can be started by “AM” command. Subsequently we start all of the activities and extract corresponding UI information. Then we
extract the UI structure features from three dimensions by introducing the concept of “Layer”. The similarity of two activities is calculated based on various structural features. Finally, we could determine the similarity of two apps based on the portion of similar activities.

**A. APP PREPROCESSING**

To start each activity explicitly using explicit intents, we insert attributes to all the activities declared in the AndroidManifest.xml file. We use Apktool [28] to decompress and repackage apps. As shown in FIGURE 3, after we get the AndroidManifest.xml file, we insert the “intent-filter” tag to each activity node, accompanied by the “category” and “action” sub tags to the “intent-filter” tag. It is noteworthy that the “category” tag must contain “android:name” attribute with its corresponding value “android.intent.category.launcher”. We repackage and sign the modified files into the “new” APK.

**B. UI DATA EXTRACTION**

In android platform, developers typically configure the user interface by defining them in either the XML files or Java code. In the implementation process of the interface, developers can directly use widgets provided by the official SDK, but also freely develop widgets for invocation. Thus it is not feasible to extract structure features based on static analysis directly.

Taking into account the scale of apps in various markets, it is also a challenge to extract UI information dynamically. In general, dynamic testing tools [30], [31] extract the UI information in the process of traversing all the components. It usually first builds a widget tree of the app, analyzes the trigger methods of each widget and then traverses the app by generating inputs. However, in the process of dynamic app automation, it is time-intensive to analyze every entry point and traverse all the components. Furthermore, the code coverage of current dynamic testing tools is relative low as suggested by previous work [32]–[35].

In this paper, we use the “AM” tool to implement the traversal of the UI. We start each activity by the command “am start”. Even though this method poses a risk of losing information when activities fail to start, the complexity and required time is much less compared to generating inputs for the execution of the entire app, which was demonstrated to be effective to traverse the UIs by previous work [21]. In the phase of widget information extraction, we use UIAutomator [36] to dump the view hierarchy. As our purpose is to extract the structure features of the UIs, we mainly focus on three attributes of each node. The “text” attribute represents the text that is displayed on the screen. The “class” attribute is the name of the class that the widget belongs to. The “bounds” attribute represents the position and the area of the rectangular region that widget controls. On account of that the arrangement of the widgets in the dumped file is outside-to-inside, we convert it into the top-to-bottom arrangement in order to facilitate the extraction of structures.

---

**FIGURE 2. Clone detection model.**

**FIGURE 3. The manifest file after adding sub tags to “activity” node.**
C. STRUCTURE FEATURE EXTRACTION

In android apps, each activity can be set to portrait mode (default) or landscape mode by app developers. We could identify the mode based on the attribute `android:screenOrientation`. In either mode, the UI is provided through a widget tree, and each widget controls a specific rectangular region in the window of an activity and can respond to user actions. The area of the rectangular region is represented as \( \text{bounds} = [(x_1, y_1), (x_2, y_2)] \), in which \( x_1 \) and \( x_2 \) are lower bound and upper bound of transverse axis respectively, \( y_1 \) and \( y_2 \) are lower bound and upper bound of longitudinal axis respectively.

Then we can extract the hierarchical structure of user interface. First, we propose a method to separate the screen by using the maximum number of non-overlapped rectangular spaces based on the basic idea that each widget must be included in a rectangular space. As shown in FIGURE 4, we finally get several rectangular spaces and we define each of them as a “Layer”. Then we merge the layers whose widgets are the same. As shown in FIGURE 5, we define these layers as an “Overlap Layer”. Particularly, we found that the overlap layer generally includes the main information of the UI. As an example, the screenshot on the left of FIGURE 4 is an news app named “BBC News”, which contains 7 layers, 3 independent layers and 1 overlap layer, then the overall hierarchy structure can be represented as \( \{7, 3, 1\} \).

1) OVERALL HIERARCHY STRUCTURE

In order to compute the overall hierarchy structure of the UI from a large amount of widgets’ information, we have implemented an XML parser to extract the value of three attributes of each widget, and then use a fast and accurate algorithm to extract overall hierarchy structure. As shown in FIGURE 6, if the activity is in portrait mode, we first take \( y_1 \) and \( y_2 \) of first widget as the lower bound and upper bound of layer 1, separately. In the recursive walk of UI information, if the ordinate range of the next widget is within this layer, then we add it to this layer and continue down the path. If not, the ordinate range is regarded as the lower bound and upper bound of the next layer. In this algorithm, the layer that contains only one widget which does not have any text and picture information will be discarded, because the information of the UI is mainly embodied in the combination display of multi filled-in widgets. After we extract the layers, we obtain the overlap layer by merging the layer as shown in FIGURE 5. The output of our algorithm is the information of layers and overlap layers. According to the experimental results, the algorithm is fast and accurate. We use \( f_1 = \{L_1, L_2, L_3\} \) to represent such characteristics, in which \( L_1 \) represents the number of layers, \( L_2 \) represents the number of independent layers, and \( L_3 \) represents the number of overlap layers. As an example, the screenshot on the left of FIGURE 4 is an news app named “BBC News”, which contains 7 layers, 3 independent layers and 1 overlap layer, then the overall hierarchy structure can be represented as \( \{7, 3, 1\} \).

1http://app.mi.com/details?id=com.iyuba.bbcnews
2) TEXT STRUCTURE

Similarly, we also extract the value of “text” within each widget. We use $f_2 = \{ C_1, C_2, \ldots, C_n \}$ to represent text structure, and $C_n$ represents the text collection of widgets in the n-th independent layer. Specifically, if the m-th independent layer is an overlap layer, then $C_m$ represents the intersection of text of all the layers belongs to the overlap layer.

3) WIDGET STRUCTURE

A widget is a visible and interactive view element of the screen. In this paper, we counted all the classes supported by the widget package in Android API 25, in which a total of 63 widgets and 7 layouts are included. By calculating the number of calls for each type of widget and layout in an independent layer, we can get a 70-dimensional feature vector $\{W_1, W_2, \ldots, W_{70}\}$, where i-th dimension represents the number of times a widget (layout) appearing in the independent layer. Particularly, if the independent layer is an overlap layer, we count only one layer which is belong to the overlap layer. In addition, if the widgets are not supported by the widget package, in other words the third-party vendor widgets, we calculate the sum of the times that they appear to generate the last dimension. Finally, a n*71-dimensional feature matrix, $f_3 = n \times \{W_1, W_2, \ldots, W_{71}\}$, is obtained to represent the widget structure.

D. SIMILARITY COMPARISON

In this paper, we measure the similarity of two Android apps based on the proportion of similar activities between them. We can determine the similarity of two activities based on comparing the extracted structural features, which is the similarity between feature vectors.

1) SIMILARITY OF ACTIVITIES

Pair-wise activity comparison is time-consuming due to the large mount of activities. We found in the experiment that
many apps have more than 10 activities whose layers are more than 5, resulting in almost 3,000 times of feature vector comparison between two apps, which will seriously affect the efficiency of clone detection. In addition, we find that many comparisons are redundant as the two activities are totally different.

Therefore, in this paper, we propose to optimize the comparison based on the overall hierarchy structure. If the structure of the two activities is quite different, it can be assumed that the two activities are not similar and there is no need to compare other detailed features. The difference of overall hierarchy structure is mainly reflected in two aspects: on one hand, the difference between the number of overlap layers is relatively large which means that the two activities are dissimilar. As we do not know how many layers are in each overlap layer in the activity, the difference of overall hierarchy structure can not be detected by independent layers. We set two thresholds in the process of preliminary comparison, if

\[ |L_{1A} - L_{1B}| > V_{\text{thres1}} \text{ or } |L_{3A} - L_{3B}| > V_{\text{thres2}}, \]

we consider that the activity A and activity B are not similar. Here, we enforce \( V_{\text{thres1}} \) to 3 and \( V_{\text{thres2}} \) to 1, which are empirically set based on our experiment.

For text structure and widget structure, since the dimensions of matrices between two activities are not necessarily the same, we calculate the percentage of the similar vectors between two matrices as their similarity. Take the case of text structure, the similarity of activity A and activity B can be calculated by

\[
SIM(f_{2A}, f_{2B}) = \frac{\sum^{m \times n}_{k=1} 2 \times SIM(C_{iA}, C_{jB})}{m + n},
\]

\[
(\forall 1 \leq k \leq m, 0 \leq n \sum^{n}_{k=1} 2 \times SIM(C_{iA}, C_{jB}) \leq 1)
\]

(1)

with the vector comparison function

\[
SIM(W_{iA}, W_{jB}) = \begin{cases} 
1 & \text{if } \min(W_{iA}, W_{jB}) \times \cos(W_{iA}, W_{jB}) \geq 0.6 \\
0 & \text{if not}
\end{cases}
\]

(4)

As plagiarists might try to evade detection by modifying text (e.g., synonym replacement), we consider that the two text structure vectors are similar if they have the contains relationship. Unlike text structure, it may be more hard for plagiarists to replace the widgets with little affecting the runtime appearance towards the user. So in this paper, we only consider the attack types by adding or deleting widgets, and we empirically set the threshold as 0.6 based on our initial experiment.

To compute the similarity of activities, we obtain the proportion of similar text vectors and the one of similar widget vectors. The thresholds are empirically set to 0.66 and 0.7 based on the experiment results, respectively. The formulas is as follows:

\[
SC(A, B) = \begin{cases} 
1 & \text{if } SIM(C_{iA}, C_{jB}) \geq 0.66 \text{ or } SIM(W_{iA}, W_{jB}) \geq 0.7 \\
0 & \text{if not}
\end{cases}
\]

(5)

Note that the thresholds in our paper are evaluated based on our benchmark dataset, as shown in section IV-A. We followed the selection method of threshold in existing researches [6], [7], decreasing the threshold gradually and calculating the false positive cases and false negative cases under each threshold. We set the best value as the threshold.

2) SIMILARITY OF APPS

Similarity of two apps is equivalent to the proportion of similar activities between them. The activity sequences of app P and Q can be represented as \( V = \{P_{A1}, P_{A2}, \ldots, P_{Am}\} \), \( U = \{Q_{B1}, Q_{B2}, \ldots, Q_{Bm}\} \), and the similarity score can be calculated by formula:

\[
SIMAPP(APP_P, APP_Q) = \frac{\sum^{m \times n}_{i=1} SC(V_{i}, U_{i}) + \sum^{n \times m}_{i=1} SC(U_{i}, V_{i})}{U + V} \in [0, 1],
\]

\[
(\forall 1 \leq k \leq m, 0 \leq n \sum^{n}_{k=1} SC(P_{Ak}, Q_{Bm}) \leq 1)
\]

(6)

In view of that some attackers might add several fake interfaces or simply copy the interfaces with core functionality of other apps, causing a big gap in the number of activities between the two apps. We identify two apps as clones when the weighted values of the two similarity scores are over the threshold. During our experiment, we empirically chose the threshold as 63%.
TABLE 2. Dataset of apps.

| Benchmark Dataset   | # of Apps | # of Clone pairs | App Category of Large-Scale Dataset | # of Apps | # of App Pairs | # of App Pairs Detected as Clones |
|---------------------|-----------|------------------|------------------------------------|-----------|----------------|----------------------------------|
| AndroZoo            | 97        | 1,892            | Picture                            | 352       | 61,776         | 16                               |
| Malicious Apps      | 43        | 53               | Read                               | 552       | 152,076        | 121                              |
| Yunshang            | 75        | 2,775            | Tool                               | 463       | 106,953        | 62                               |
| Obfuscated Apps     | 30        | 20               | Video                              | 410       | 83845          | 838                              |
| Total               | 245       | 4740             | Total                              | 1777      | 404,650        | 1037                             |

IV. EXPERIMENT RESULTS

In this section, we have implemented an automated prototype system and then conducted measurement study to evaluate the effectiveness and scalability of our approach. Our study is mainly focused on the following two research questions:

RQ1 How effective is our approach in detecting app clones? How does our approach compare with existing tools? As we aim to apply our approach to detect app clones from the large dataset, it is important to evaluate the effectiveness of our approach on detecting different types of clone attacks.

RQ2 How efficient is our approach? Could it be applied to market-level detection? Our approach need to start all activities of an app to extract runtime UI birthmarks, which takes more time than extracting static app birthmarks. Therefore, we need to evaluate the scalability and efficiency of our approach on a large dataset.

Our experiments are conducted on a Linux machine with 3.5GHz i3-4150 CPU and 8GB of RAM. During the preprocessing, we use Apktool [28] to decompress and repackaged apps. In addition, we use the signature tool “Signapk” [37] to resign the app. We use a collection of TPLs based on previous work [29]. In other phases we implemented our approach in Python and shell script. Note that app preprocessing and the feature extraction only need to be performed once.

A. DATASET

In order to answer RQ1, we need to harvest a comprehensive dataset that covers different types of app clones which leverage different obfuscation techniques. In this paper, we take advantage of existing efforts, and use several sets of manually labeled real-world apps to build our benchmark dataset. Eventually, as shown in TABLE 2 we build a dataset of 245 apps, which contain 4,740 clone pairs and can basically contain the three types of clone attacks as described in Section IV-B.2.

- AndroZoo Dataset [38], an academic effort focused on compiling a large-scale dataset of APKs. We randomly select a set of repackaged apps.
- Dataset from Daniel et al. [39], a sample set of malicious apps. We randomly select a set of repackaged apps.
- Yunshang Dataset, a set of apps which are downloaded manually from the app market. Note that “Yunshang” apps are developed using the same framework, some of these apps have dissimilar functional code and some have dissimilar UI code, while the UI structures of them are very similar
- Obfuscated App Dataset, 10 original testing apps which are developed by the first author of the paper. We then use Allatori [40] which is a widely used code obfuscation tool, and following Luka et al. [7] to develop a UI obfuscation tool that modifies the activity transition graphs of an app, to generate 20 obfuscated apps.

In order to answer RQ2, We build a large scale dataset of apps. These apps are downloaded from two Android third party markets [41], [42], a total of 1,777 apps contains four categories. As app clones generally appear in the same category, the apps in the four categories are respectively pairwise compared, consequently these 1,777 apps can compose 404,650 app pairs as shown in TABLE 2.

B. RQ1: EFFECTIVENESS OF OUR APPROACH

1) DETECTION ACCURACY

We first evaluate the accuracy of our approach on detecting clone apps. We compare the apps pair-wisely (in total 23,025 pairs), FIGURE 7 shows the distribution of the similarity score among the app pairs. We measured the performance of our system at different thresholds. As shown in FIGURE 8, with the threshold of 0.63, 4,720 pairs were successfully detected out of 4,740 clone pairs and 3 pairs were false positives, which means our false positive rate FPR=0.02% and false negative rate FNR=0.42%. We analyze the reasons leading to false positives and false negatives by examining the screenshots of the running activities and found that:(1) each app among the 3 false positive pairs has only 1 activity and the text between them are similar. (2) Most of the false negative apps belonged to AndrooZoo dataset, 17 of the cases fail to run and leading to either black screen or no-information UI. The others are apps who have only one activity without text. The results show that our approach is obfuscation-resilient and performs well in detecting different types of clone attacks.

2) COMPARE WITH STATE-OF-THE-ART APPROACHES

We then compare our approach against available implementation of three state-of-the-art works, namely FSquaDRA [25], SimiDroid [43] and a tool reimplemented of the approach proposed in research [8], covering respectively resource-based, code-based and UI content-based similarity analysis.
FSquaDRA is a static repackaging detection approach based on the similarity of resource files. The tool used 18 file types as a feature vector and detect app repackaging by considering separately different types of files. Furthermore, it indicates that Overlap performs better than other classifiers. In this section, we mainly focus on the similarity score of “Overlap res all”, which calculate the similarity of files located under “res/” folder. We set the threshold as 0.8 according to previous work [44], [45].

b: SIMIDROID
SimiDroid is a reusable tool for detecting similar Android apps at the method level. The similarity of SimiDroid is calculated based on four metrics and can be explained at different levels which can be further enriched via plugin implementations. In this paper, we set the value of “plugin name” in the plugin file as “METHOD”, to get the similarity of functional code. We set the threshold as 0.9 according to previous work [43].

c: REIMPLEMENTED TOOL
This tool is propose by [8], to compare the GUI similarity among Android apps by calculating similarity of text elements and image elements respectively. It set different weight for proposed 8 different types of features. We set the threshold as 0.8 according to their experiment result [8].

We run these three tools on our benchmark dataset, and the detection result is shown in TABLE 3. The three tools perform well at detecting repackaged apps in the dataset of 77150 VOLUME 8, 2020
AndrooZoo and malicious apps, the false negative cases are game apps whose code are mostly native code, the tools can not extract efficient static birthmarks to calculate the similarity. For the dataset of Yunshang, some of the app pairs have similar visual but different hashes of the resource files, and the resource files have been substantially changed (e.g., new folders were introduced; images were replaced), which leads to outliers detected by FSquaDRA and reimplemented tool. Clone apps detected by SimiDroid is much fewer, that is because most of the plagiarists will modify parts of the functional code to avoid clone detection. For the dataset of obfuscated apps, FSquaDRA and reimplemented tool are resistant to code obfuscation while SimiDroid is not, because the tool Allatori [40] do not modify the UI code, but replace the variable in the functional code and even change the code structure. Worse yet, all the three detection tools are ineffective in detecting clone apps which use UI obfuscation. The UI obfuscation tool [7] build new activities and add new element (e.g., buttons) into existing activities, to modify the activity transition graphs of an app, thus making many changes to the functional code and UI code of original apps. The evaluation results suggest that comparing with existing app clone detection tools, our approach is obfuscation-resistant and is effective on detecting app repackaging and other sophisticated clone attacks.

C. RQ2: EFFICIENCY EVALUATION

In this section, we applied the prototype system to a large scale dataset of apps. During the phase of app preprocessing and data extraction, we use two Android smartphones (Nexus 5) to start the activities for each app. The system deployed on Linux is responsible for assigning extraction tasks to an idle phone. As the method of data extraction in this paper does not involve component analysis and input-event construction, the relationship between the time of extracting widgets’ information and the number of activities is linearly related. For the activity that started successfully, the time of data extraction is set to 5 seconds in this paper. FIGURE 9 shows the distribution of number of successfully started activities for each app, and we can observe that majority of the apps successfully start less than 40 activities. Therefore, the time needed to dump the widgets’ information from each app is about 4 minutes on average. The results show that when the system processes large-scale apps, the time of data extraction can be controlled according to the number of test terminals.

For the dataset of 404,650 app pairs (1,777 apps), the time from birthmarks generation to clone sets detection takes about 6 hours, an average of 0.053 seconds to compare an app pair. Note that we propose to optimize the comparison based on the overall hierarchy structure, which greatly decrease the times of comparison, making the efficiency of our approach be acceptable.

Eventually, the prototype system successfully detect 1037 clone app pairs, including 272 distinct apps. We upload these apps to VirusTotal [46] to determine whether they are malicious or not. In addition, we evaluate the ability of our approach in identifying different clone types by manually analysis.

(1) About 83% of the 272 clone apps detected from our experiment are reported as malware by at least one engine of VirusTotal. Most of them are adware and Trojan horses, the others are spyware.

(2) Most of the clones we detected are repackaged apps. Nevertheless, we found two clone pairs in video category belonging to UI clone attacks, the apps are designed for different requirements with corresponding video resources and signed with different keys. In addition, we also found two clone sets in tool category belonging to functional clone attacks. The apps have similar functionality used for Wi-Fi crack and system root, but the number and attributes of widgets are not similar. The results show that our approach can detect different types of clone attacks.

V. DISCUSSION

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

A. THREATS TO VALIDITY

First, our approach poses a risk of losing information when activities fail to start. As we aim to apply our approach to detect app clones from market-level dataset, we need to cross the limitation of existing automated testing techniques, which is too time-consuming to extract enough information and the UI coverage is relatively low. Therefore, we propose to start each activity explicitly by using intents, to extract UI information for software birthmark generation. The experiment results show that our approach is effective to achieve the balance among UI coverage, efficiency and the performance. Second, our approach inherits the drawbacks of existing approaches. Though the thresholds we selected in our paper was proven to be quite effective during our evaluation, they may be not appropriate regarding to other datasets and will introduce false negatives and false positives. While not perfect, the precision of our approach could enable markets to detect most of the clone attacks. Besides, markets might be to combine various detection methods to achieve a trade-off.

2Examples of app clones we detected as UI clone attacks [47]–[51]
3Examples of app clones we detected as functional clone attacks [52]–[56]
B. ORIGINAL APP DETERMINATION

Although clones can be found successfully by our system, in most of the cases it is unable to define which app is the original app. Especially when there is a set of clones, things will get more complicated. Some of the previous work [5], [57] have proposed some heuristic solutions, such as checking the submission time of the app, the popularity of the app, and the size of APK, etc. However, none of these solutions are perfectly sound and it is easy for hackers to evade the detection.

For future work, we would like to conduct more in-depth studies on mining user-interface features/behaviors of plagiarism apps, to find a solution of identifying original apps.

VI. RELATED WORK

A. APP CLONE AND FAKE APP DETECTION

There has been a lot of researches on the clone detection for android apps. Most of the previous approaches are based on static features extracted from code [6], [57]–[60]. Zhou et al. [58] leveraged the fuzzy hashes of each method to generate the fingerprint. Subsequently, Zhou et al. [57] split an app’s code into primary and non-primary modules, then a semantic feature fingerprint for each primary module can be extracted to address the problem of detecting “piggybacked” apps. JustApp [59] leveraged feature hashing to detect code reuse in Android apps. Wang et al. [6] presented WuKong, a two-phase approach which combines coarse-grained detection by comparing light-weight static features and fine-grained detection by comparing more detailed features. To eliminate the impact of third-party libraries, a number of tools were proposed [61], [62] to filter TPLs efficiently.

To capture the high-level semantic information of the code, Androguard [63] was proposed to compare the similarity between apps based on control flow graph. Desnos and Geoffroy [16] developed DNADroid to check app repackaging by program dependency graph. Lastly, they apply a filter to remove unlikely clones, before comparing the rest of the PDG pairs that passed the filter using a subgraph isomorphism. AdDarwin [64] cut PDGs into connected components and extract semantics vector of each component. Furthermore, the semantic vectors can be used to detect external libraries. Chen et al. [27] extracted the methods and constructed a 3D-control flow graph called 3D-CFG to get the centroid. They then leveraged the centroid to group similar apps together.

Differing from code similarity based approaches, several studies extract fingerprints from other files within apps to detect clones. FSquaDRA [44] was implemented based on the comparison of the resource files, this approach is resilient to code obfuscation, but will be affected by some changes in the resources. A further study [25] was proposed to demonstrate that a very low proportion of identical resource files in two apps is a reliable evidence for repackaging. Viewdroid [65] and MassVet [66] statically analyzed the UI code to extract a graph that expresses the user interface state and transitions.

However, all these approaches that were focused on static fingerprints may be vulnerable to advanced obfuscation techniques and clone attacks. Recent years, researches mainly focused on detection techniques. Charlie et al. [21] proposed that they are the first to use runtime UI birthmarks to Android app clone detection. Yury et al. [25] extracted UIs from apps and analyzed the extracted screenshots to detect impersonation. Luka et al. [7] extracted features from the attribute of text components and image components to calculate similarity of UI. Malisa et al. [9] detected mobile app spoofing attacks which can be regarded as an advanced partial UI clone attack by leveraging user visual similarity perception, they conducted an extensive online user study and the result told how likely the user is to fall for a potential clone attack. Our approach differs from them in that our birthmark information is extracted from interface structure information.

B. SOFTWARE PLAGIARISM DETECTION

Hyun-il et al. [67] leveraged stack pattern based birthmarks which requires the source code. Myles and Christian [68] statically analyzed executables and proposed op-code level k-gram based static birthmarks. He et al. [69] used fuzzy hashing method to measure the code clones in the smart contract ecosystem. Haruaki et al. [70] proposed four types of static code-level birthmark to detect theft of Java programs. There are also dynamic software birthmarks based methods, including system call based birthmarks [71], [72], core value based birthmarks [73], [74] and dynamic API based birthmarks [75], [76].

C. UI INFORMATION MINING

Besides mobile security, information of Android app GUIs has also been studied for software engineering. Recent researchers extract screen semantics from Android app GUIs and task flow from Android app GUI layouts, to mine human-generated app traces [77], [78], add natural language interfaces [79]–[81], identify the inconsistence between intention and app behaviors [82], [83], detect aggressive mobile advertisements [84]–[86], and build conversational bots [87], [88]. Thomas et al. [77] introduced an automatic approach for generating semantic annotations for mobile app UIs, which could be used to develop new data-driven design applications and enable efficient flow search over large datasets of interaction mining data. Siva and Christoph [78] designed P2A, a tool used for automate human perception and manual data entry tasks by developers. Biplab et al. [80] presented ERICA, a system that takes a scalable, human-computer approach to interaction mining existing Android apps without the need to modify them in any way, which could be further used for usability testing and online app indexing and searching.

VII. CONCLUSION

In this paper, we presented a novel approach to detect Android app clones based on the similarity of UI structure. We extract UI structure features from view hierarchy information to generate the dynamic software birthmark. The result shows that our approach can effectively detect different types of clone attacks, including repackaging, functional cloning and
W. Zhou, Y. Zhou, and X. Jiang, “Detecting mobile application spoofing,” in Proc. 3rd ACM Conf. Data Appl. Secur. Privacy (CODASPY), 2013, pp. 185–196.

W. Zhou, Y. Zhou, and X. Jiang, “Detecting mobile application spoofing attacks by leveraging user visual similarity perception,” in Proc. 2nd ACM Conf. Data Appl. Secur. Privacy, 2012, pp. 317–326.

S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, “Juxtap: A scalable system for detecting code reuse among Android applications,” in Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment. Berlin, Germany: Springer, 2012, pp. 62–81.

W. Yang, X. Xiao, D. Li, H. Li, H. Wang, Y. Guo, and T. Xie, “Security analytics for mobile apps: Achievements and challenges,” J. Cyber Secur., vol. 1, no. 2, pp. 1–14, 2016.

Z. Ma, H. Wang, Y. Guo, and X. Chen, “LibRadar: Fast and accurate detection of the common libraries in Android apps,” in Proc. 38th Int. Conf. Softw. Eng. Companion (ICSE), 2016, pp. 653–656.

L. Li, T. Riom, T. F. Bissyandé, H. Wang, J. Klein, and L. T. Yves, “Revisiting the impact of common libraries for Android-related investigations,” J. Syst. Softw., vol. 154, pp. 157–175, Aug. 2019.

(2017). AndroGuard: Reverse Engineering, Malware and Goodware Analysis of Android Applications. [Online]. Available: https://code.google.com/p/androguard/

J. Crussell, C. Gibler, and H. Chen, “Scalable semantics-based detection of similar Android applications,” in Proc. 18th Eur. Symp. Res. Comput. Secur. (ESORICS), 2013, pp. 1–21.

F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, “ViewDroid: Towards obfuscation-resilient mobile application repackaging detection,” in Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw. (WiSec), 2014, pp. 25–36.

K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, “Finding unknown malware in 10 seconds: Mass vetting for new threats at the Google-play scale,” in Proc. 24th USENIX Secur. Symp. (USENIX), 2015, pp. 659–674.

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., vol. E91-D, no. 9, pp. 2323–2332, Sep. 2008.

G. Myles and C. Collberg, “K-gram based software birthmarks,” in Proc. ACM Symp. Appl. Comput. (SAC), 2005, pp. 314–318.

N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, “Characterizing code clones in the Ethereum smart contract ecosystem,” 2019, arXiv:1905.00272. [Online]. Available: http://arxiv.org/abs/1905.00272

H. Tamada, M. Nakamura, A. Monden, and K.-I. Matsumoto, “Design and evaluation of birthmarks for detecting theft of java programs,” in Proc. IASTED Conf. Softw. Eng., 2004, pp. 569–574.

X. Wang, Y.-C. Jhi, S. Zhi, and P. Liu, “Behavior based software theft detection,” in Proc. 16th ACM Conf. Comput. Commun. Secur. (CCS), 2009, pp. 280–290.

X. Wang, Y.-C. Jhi, S. Zhi, and P. Liu, “Detecting software theft via system call based birthmarks,” in Proc. Annu. Conf. Comput. Secur. Appl. Conf., Dec. 2009, pp. 149–158.

Y.-C. Jhi, X. Wang, X. Jia, S. Zhi, P. Liu, and D. Wu, “Value-based program characterization and its application to software plagiarism detection,” in Proc. 33rd Int. Conf. Softw. Eng. (ICSE), 2011, pp. 756–765.

F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhi, “A first step towards algorithm plagiarism detection,” in Proc. Int. Symp. Softw. Test. Anal. (ISSTA), 2012, pp. 111–121.

D. Schuler, V. Dullmeier, and C. Lindig, “A dynamic birthmark for java,” in Proc. 22nd IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE), Nov. 2007, pp. 274–283.

H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.-I. Matsumoto, “Dynamic software birthmarks to detect the theft of windows applications,” in Proc. Int. Symp. Future Softw. Technol., vol. 20, 2004, pp. 1–6.

T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, “Learning design semantics for mobile apps,” in Proc. 31st Annu. ACM Symp. User Interface Softw. Technol. (UIST), 2018, pp. 569–579.

S. Natarajan and C. Csalnner, “P2A: A tool for converting pixels to animated mobile application user interfaces,” in Proc. IEEE/ACM 5th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft), May/Jun. 2018, pp. 224–235.

K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, “Machine learning-based prototyping of graphical user interfaces for mobile apps,” 2018, arXiv:1802.02312. [Online]. Available: http://arxiv.org/abs/1802.02312

B. Deka, Z. Huang, and R. Kumar, “ERICA: Interaction mining mobile apps,” in Proc. 29th Annu. Symp. User Interface Softw. Technol. (UIST), 2016, pp. 767–776.

T. J.-J. Li, A. Azaria, and B. A. Myers, “SUGILITE: Creating multimodal smartphone automation by demonstration,” in Proc. CHI Conf. Hum. Factors Comput. Syst., May 2017, pp. 6038–6049.

S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu, and J. Lu, “DeepIntent: Deep icon-behavior learning for detecting intent-behavior discrepancy in mobile apps,” in Proc. ACM SIGSAC Conf. Comput. Secur. Commun., Nov. 2019, pp. 2421–2436.

H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong, “Understanding the purpose of permission use in mobile apps,” ACM Trans. Inf. Syst., vol. 35, no. 4, pp. 1–40, Aug. 2017.

F. Dong, H. Wang, L. Li, Y. Guo, T. F. Bissyandé, T. Liu, G. Xu, and J. Klein, “FraudDroid: Automated ad fraud detection for Android apps,” in Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), 2018, pp. 257–268.

T. Liu, H. Wang, L. Li, X. Luo, F. Dong, Y. Guo, L. Wang, T. F. Bissyandé, and J. Klein, “MadDroid: Characterising and detecting devisous ad content for Android apps,” in Proc. World Wide Web Conf., 2019, pp. 1988–1999.

T. Liu, H. Wang, L. Li, G. Bai, Y. Guo, and G. Xu, “DaPanda: Detecting aggressive push notifications in Android apps,” in Proc. 14th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE), Nov. 2019, pp. 66–78.

T. J.-J. Li and O. Riva, “Kite: Building conversational bots from mobile applications,” in Proc. 26th Annu. Int. Conf. Mobile Syst., Appl., Serv., June 2018, pp. 96–109.

M. Ghazvininejad, C. Brockett, M.-W. Chang, B. Dolan, J. Gao, W.-T. Yih, and M. Galley, “A knowledge-grounded neural conversation model,” in Proc. 32nd AAAI Conf. Artif. Intell., 2018, pp. 5110–5117.

YANGU HU received the bachelor’s degree in applied physics from the Beijing University of Posts and Telecommunications, Beijing, in 2012, where he is currently pursuing the Ph.D. degree with the School of Cyberspace Security. His research interest includes mobile application security and privacy.

GUOSHENG XU (Member, IEEE) received the Ph.D. degree in information security from the Beijing University of Posts and Telecommunications, China, in 2008. Since July 2008, he has been a Lecturer with the School of Cyberspace Security, Beijing University of Posts and Telecommunications. In September 2019, he was a master’s Supervisor. His current research interests include machine learning, intelligent software security, and advanced cryptography.
BOWEN ZHANG is currently pursuing the bachelor’s degree with the Beijing University of Posts and Telecommunications. His research interests include blockchain security and mobile security.

KUN LAI is currently pursuing the bachelor’s degree with the Beijing University of Posts and Telecommunications. His research interest includes mobile security.

GUOAI XU received the Ph.D. degree in signal and information processing from the Beijing University of Posts and Telecommunications, China, in 2002. He is currently an Associate Director of the National Engineering Laboratory of Security Technology for Mobile Internet, School of Cyberspace Security, Beijing University of Posts and Telecommunications. His research interests include software security and data analysis. He received the Title of Professor in 2011.

MIAO ZHANG received the Ph.D. degree from the Beijing University of Posts and Telecommunications, China, in 2007. He is currently an Assistant Professor with the Beijing University of Posts and Telecommunications. His research interests include software security and network security and countering.