# SUIDroid: An Efficient Hardening-Resilient Approach to Android App Clone Detection

Fang Lyu[†], Yapin Lin[*†], Junfeng Yang[†], Junhai Zhou
College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
[†]Hunan Provincial Key Laboratory of Dependable Systems and Networks, Hunan, Changsha, China
[*]Email: yplin@hnu.edu.cn

***Abstract*—The huge benefit of mobile application industry has attracted a large number of developers. However, application cloning is becoming a serious threat to the entire Android ecosystem, as it not only compromises the security and privacy of the app users but also robs app developers' revenue. As massive approaches have been proposed to address this issue, plagiarists also fight back through hardening their malicious code with the help of commercial packers. Previous work either not considers the hardening issue or just simply cracks the entire packed app, which may threat the security of legitimate packed app. In this paper, we propose *SUIDroid*, a novel app clone detection approach that can accurately detect the packed clone app without compromising the app hardening strategy. Another advantage of our detection approach is the accuracy. We propose a counting based technique to effectively filter the *noisy-layout* that can cause deviation. We evaluate our approach on two sets of app dataset. The experimental results on 160 packed malware samples demonstrate that SUIDroid is resilient to app hardening, and the evaluation on more than 100,000 real-world apps shows the efficiency of SUIDroid in real-world scenario.**

***Keywords—Android; Clone detection; User interface; Noisy-layout; Hardening resilient***

## I. INTRODUCTION

The worldwide smartphones grew 13.0% year over year in 2015 and Android dominated the mobile system market with a share of nearly 83% [1]. Mobile applications (*apps* for short) play a significant role in the Android ecosystem. And the gross value of mobile application industry will reach $143 billion in 2016, according to the forecast by Google [2].

However, app cloning [4] may let the app developers lose their potential revenue. Given the openness of Android market, plagiarists can easily obtain a popular app's installation package (i.e., the .apk file) and disassemble it with the help of some popular decompiling tools [5, 6]. After that, attackers can modify the logic of the original app (e.g., crack paid apps to bypass payment function, modify the advertisement libraries or even insert various malicious functions) and then repackage the app with a new signature before republishing the clone app into other third-party markets. By "copying" the user interface (UI for short) of a popular app, the clone app can easily deceive users and utilize the popularity of the original app to distribute itself widely [8, 14].

Both academia and industry have done a lot of work to address this issue. The researchers try to build app clone detection system for Android app markets [8-19] to filter out illegal apps. The most core of their work is to construct a novel *birthmark*, which is defined as the unique characteristic to identify an app [11], for similarity comparison. According to the type of *birthmark*, the detection techniques can be divided into two categories: *code-based* techniques [13-19] and *UI-based* techniques [8-12]. *Code-based* methods focus on detecting the code reuse in repackaged apps, while *UI-based* methods analyze the similarity of app's user interface.

On the other hand, in the industry, the commercial security companies [21] focus on the security of single app and adopt various code protection techniques (a.k.a., app packing or app hardening) to encrypt or obscure the original app [20]. The packed app can effectively resist reverse engineering and code disassembling.

It is encouraging that several state-of-the-art app clone detection systems [9, 14] have claimed to achieve accuracy and scalability in real-world scenario. However, the commercial app packing, which is intended for protecting benign apps from being disassembled and repackaged at first [21], is now becoming an effective practice for illegal apps to thwart current security detections. So far, about 10.0% of Android malwares are packed, and unfortunately, current app clone detection techniques, no matter *code-based* or *UI-based*, do not pay much attention to the app packing issue, thus are not able to identify the clone apps among the packed apps [20].

Although several approaches [20, 21] were proposed that can effectively disassemble the encrypted code and even rebuild the packed app, these schemes also provide attackers with the ability to crack the legitimate app that protected by the app packing. Robbing Peter to pay Paul is not a good choice.

In this paper, to address the problem of packed app clone with no compromise, we proposed SUIDroid, a novel app clone detection system that can accurately detect the packed clone app without cracking the entire packed app. Our approach is motivated by the following two observations: First, not all files in the app are packed. Actually, only the Dalvik executable (DEX) file is encrypted or obscured by app hardening services. The DEX file contains all the source codes

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.103

that implement the functionality of an app. While other resource files (e.g., icons, images, layout XML files) are stored in a separate folder and unpacked. Second, these unpacked files contain all the characteristics of app's UI. Besides the source code, these UI characteristics also maintain similarity between the clone app and the original app, as the clone app always need to make itself "look like" the original app to deceive users.

Specifically, SUIDroid proposes an *UI-based* birthmark called *Schema Layout* (the 'S' in SUIDroid) to identify both packed and normal apps. Schema layout is a special layout document that subsuming the complete layout characteristics of an app. These characteristics are extracted from separate layout files and organized in a tree structure. The resulting tree structure has the main advantage of retaining the specifics of all the enclosed layout files, while guaranteeing a compact representation. Our evaluation results demonstrate that SUIDroid has the following advantages:

**(a) Non-Aggressive Hardening Resilient:** The elements in *schema layout* are all gathered from the layout files, which are not affected by app hardening. Moreover, SUIDroid can obtain these layout files without completely cracking the packed app, as they are stored separately. Thus, our approach achieves the resilience to app hardening techniques in a non-aggressive way.

**(b) Competitive Accuracy:** *Schema layout* contains and only contains the most representative UI features of the app. We propose a novel approach to effectively filter the non-core UI features which can reduce the distinctiveness of birthmark. Since the potential noisy features are eliminated, our approach can get a desirable accuracy.

**(c) Efficient Similarity Comparison:** As the UI features of apps can be represented by the XML document, we simplify the similarity comparison among apps into searching homologous documents, which can be addressed with very high efficiency [7]. Our document-based comparsion method provides a competitive alternative to the traditional structure-based (e.g., subgraph isomorphism [8, 19]) comparisons.

In summary, our paper makes the following contributions:

- We construct a new *UI-based* birthmark, called *schema layout*, for Android apps. specially, it only relies on the features extracted from the layout XML files while still achieves enough accuracy.
- We propose SUIDroid, a novel Android app clone detection system that based on *schema layout*. SUIDroid is able to detect the packed apps without cracking the entire app and maintains both accuracy and efficiency at the same time.
- We implement a prototype system and evaluate the packing resilience of SUIDroid on 40 real-world packed app samples collected from SandDroid [29]. The experimental results show that SUIDroid is able to handle the app that packed by current mainstream commercial app packers.
- We evaluate our approach on over 100,000 real-world apps that crawled from 6 different Android markets. It turns out that 2%-14% of apps are clone cases. We also evaluate the accuracy of SUIDroid on a well-known clone app dataset [9]. The false positive rate is only 0.06%. The results demonstrate the efficiency and effectiveness of SUIDroid.

The rest of this paper is organized as follows. Section 2 describes our key ideas and the scope of our approach. The implementation of SUIDroid is presented in detail in Section 3. Evaluation is presented in Section 4, followed by related work in Section 5. Finally, we conclude the work with Section 6.

## II. DESIGN

In this section, we give out the key ideas behind our approach of overcoming the challenges to the performance of SUIDroid. At last, we claim the scope of our paper.

### A. Challenges and Strategies

Through analysis, we found that current mainstream app hardening (i.e., packing) services and app developers all only pay attention to the security of the source code (i.e., the DEX file) in app, while left the layout files not encrypted. Thus, our detection approach tries to avoid conflicts with commercial hardening by proposing a new birthmark that only relies on the layout XML files to identify the app. In this case, there are two challenges SUIDroid needs to overcome:

**(1) How to build an accurate birthmark with limit resources?**

We construct a new *UI-based* birthmark, called *schema layout*, which only uses the features extracted from the layout XML files to accurately characterize the app. Below is the formal definition of *shcema layout*.

**DEFINITION 1.** *A schema layout is a prototype XML document subsuming the most relevant layout features of the layout files within a single app, and*

- *The elements and attributes in schema layout are extracted from the layout files through static analysis.*
- *The relative relationships (e.g., including, containing and sequence) among elements in original layout files are retained in schema layout.*
- *The homogeneous subparts in different layout files is merged in schema layout. While the noisy-layout are not involved in the generation of schema layout.*

*Schema layout* is distinctive enough to identify the Android apps and we leave the detailed analysis and construction process in next Section.

The key obstacles that hinder our approach from achieving a higher accuracy is the *noisy-layout*. The *noisy-layout* refers

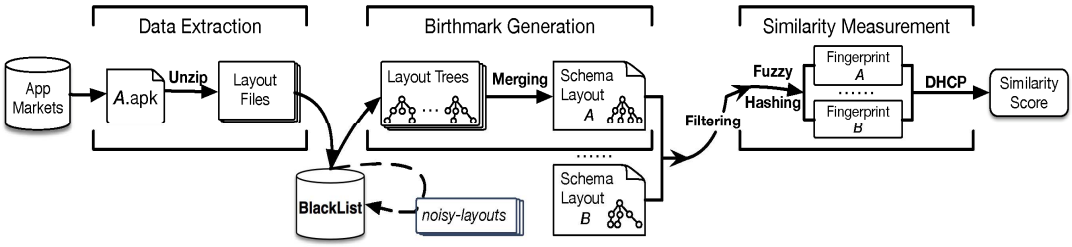


Fig. 1. An overview of SUIDroid

to two types of layout files: **(*NL-1*)** the external layout files that included by the third-party libraries and (***NL-2***) the extra layout files imported by plagiarists. We proposed a counting based technique, called *blacklist*, to filter the *noisy-layout*. Inspired by [14], *blacklist* identifies the *noisy-layout* by analyzing the frequency of the layout file in different apps:

- Within a large enough app dataset, *NL-1* are the layout files of which the frequency exceeds an upper threshold, because third-party libraries are used by many different apps.
- Inside the clone app couple (app and its clone version), the native layout files are copied once, while *NL-2* only exist in the repackaged app. Thus, the local frequency of *NL-2* in the couple is half of the native layout files.

**(2) How to achieve scalability in real-world scenario?**

There are two challenges that prevent *UI-based* detection systems from achieving scalability and efficiency.

**Challenge 1** (***C1***): The number of apps in real-world markets has reached millions and is increasing every day.

**Challenge 2** (***C2***): The number of the layout files in single app is also very impressive. Our evaluation found that each app contains 25.5 layout files on average.

We achieve our efficiency in two ways:

- To overcome *C1*, we designed a pre-filter to avoid the redundant comparisons among independent apps and then adopt the CTPH [7] hash algorithm to calculate the similarity between apps. The most prominent quality of hash algorithm is the high processing speed.
- To overcome *C2*, we extract layout features from all layout files to build an app-level birthmark, rather than a set of birthmarks [11, 12] over separate layout files. Our methods can effectively avoid the expensive comparisons among the individual layout files.

### B. Scope and Assumption

In this paper, our purpose is to detect repackaging Android app pairs, but not to identify which is the original one and which is the cloned one. We only focus on the Android apps whose layout features are defined in layout XML files. Apps with no or very few layout files, such as web apps, games based on third-party engines and background apps with only services are out of the scope of our paper. All of the apps used in our evaluation are not paid apps, so that we could have crawled enough apps to simulate the real world scenario.

## III. IMPLEMENTATION

### A. Overview of SUIDroid

We have implemented SUIDroid with 2200 lines of Python code and 500 lines of Shell code. Fig. 1 shows the overall architecture of SUIDroid.

The whole system consists of three successive parts: data extraction, birthmark generation and similarity calculation. The first task of our approach is to gather the separate layout files. Before building the birthmark, *blacklist* is used to filter the *noisy-layout*. We extract *layout trees* [11] from layout files and then merge these trees to generate schema layout. Finally, we adopt the DHCP algorithm to calculate the similarity between two *schema layouts*.

### B. Data Extraction

In SUIDroid, we need to extract app's UI features, more specifically, the layout features to characterize Android apps. Our birthmark, schema layout, that relies on the layout features is reliable and representative because:

- Repackaged app usually tries to keep the "look-and-feel" similar to original apps to deceive users.
- The layout features are independent from the source code of apps. Hence, the app hardening will not hinder us from extracting layout features from packed app.
- The speed of extracting layout features is very fast because we only need obtain the layout files from APK file, rather than disassembling the entire app.
- It takes the plagiarists' great efforts to completely re-implement an existing layout, because any slight modification of a layout structure can easily mess up the user interface.

A major drawback of our approach is that our layout-based birthmark can be easily tainted by *noisy-layout*. We proposed a counting based filtering technique, called *blacklist*, to address this issue. In addition, we employ Apktool [5] to reverse engineer the APK files. The whole data extraction phase can be broken down into two steps:

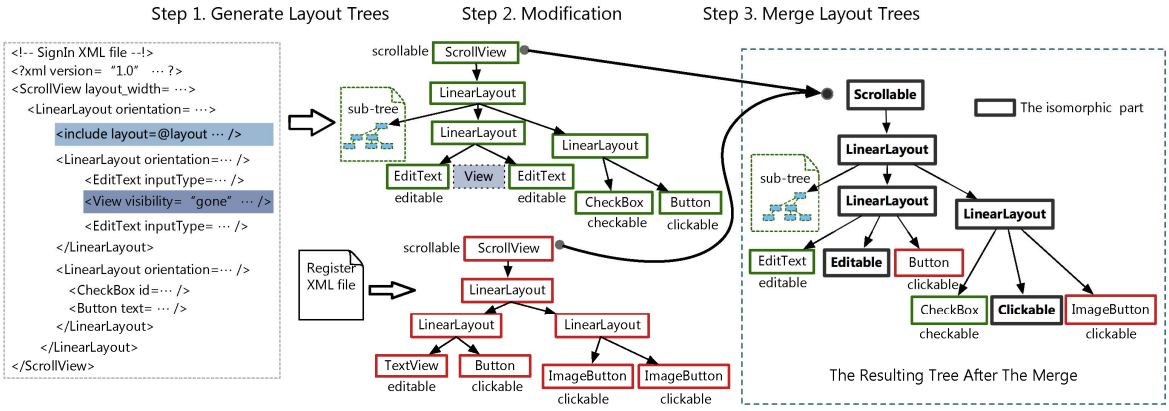


Fig. 2. An example of generating schema layout

**Step 1: Obtain layout files.** The user interface of Android apps is usually defined in layout XML files under the */res/layout* directory. Developers can also build apps UI dynamically by java code, but this method is not a common practice and not recommended by the Android official [15]. For each app, we extract all the layout files. There are several apps packed specially to disable the reverse tools. In this case, we apply the "*unzip*" command together with *AXMLPrinter* (a format conversion tools that can converts a binary file into a readable file) to extract layout files.

**Step 2: Filter *noisy-layout*.** These obtained layout files can be classified into 2 general groups: internal layout files that created by app's developers and external layout files (a.k.a., *noisy-layout*) that imported by third-libraries or attackers. The goal of this step is to filter the *noisy-layout* and we apply *blacklist* to do this job. The *blacklist* maintains a database which stores the fuzzy hash fingerprint of all identified *noisy-layout* and can update automatically. To find out the external layout files in the app, we match the hash value of every layout file in the app with the hash value stored in the database. After match, we update the frequency of these files and drop the suspicious layout files only if whose frequency exceeds the upper threshold. For example, we find out that the layout file *alipay.xml* exists in more than 700 independent apps (among 15K apps). We can determine that this layout is a typical *noisy-layout* that belongs to a well known third party library (i.e., AlipaySDK) and not created by the original app developers.

### C. *Birthmark Generation*

App birthmark is a unique characteristic of a certain app that can be used to determine the identity of app. Birthmark can be divided into two categories: *code-based* birthmark and *UI-based* birthmark. In this paper, we proposed schema layout, an UI-based birthmark that relies on app's layout features and used it for clone detection. Schema layout is a XML file that contains the layout features of an entire app. These features are extracted from separate layout files and organized according to a predefined schema document.

The task of this phase is to construct a birthmark by leveraging the layout files that filtered by the last phase. The detailed steps of generating schema layout are as follows:

**Step 1: Generate layout trees.** The layout features of apps are separately defined in layout XML files. To simplify the next merge operation, we generate a layout tree over each layout file. A node in layout tree represents an element in original XML files and the hierarchical relationship among nodes is corresponding to that in layout file. The leaf nodes in layout tree present specific visual elements (e.g., button, image, text, etc.) in user interface, while the non-leaf nodes determine the position and size of these visual elements. There is a special kind of element in layout files that can import other layout files. Hence, we first build a layout tree over the included file and then replace the corresponding statement nodes in original layout tree.

**Step 2: Modify layout trees.** We need modify the original layout trees to (a) filter the useless elements and attributes and (b) eliminate the ambiguity between the layout file and the user interface. There are many elements in layout files that make no difference to the actual user interface, but may affect the accuracy of our birthmark. This is because the hackers would like to add many invisible elements hoping to bypass the *UI-based* detection. Thus, we need remove the nodes with zero width and height attribute or whose visibility is set to "invisible" in layout trees. Some attributes of elements are also useless in our birthmark. For example, the plagiarists can change the "text" attribute from 'username' to 'login id' or just translate the text into another language. So we only remain the attributes that are critical to the appearance of elements and difficult to be modified. Dedicated attackers may try to modify the position and attributes of elements at the same time to keep the same user interface to evade our detection. We drop the position related attributes and adjust the order of the nodes at the same level in the layout tree according to its position in the

screen, as so only the sequence of node in layout tree can determine the position of element in user interface.

**Step 3: Merge layout trees.** After building layout trees over the separate layout files, we merge these trees together to generate a final matching tree for the entire app. Considering the accuracy and efficiency, we not only need reduce the number of the redundant nodes as far as possible, but also need retain the position relationship among nodes in the app-level tree. In summary, the output of this step is a minimal app-level tree that can match all separate layout trees conducted by last operation. We start merging two layout trees from the root node and then layer down. For the isomorphic parts in two layout trees, we merge these nodes and only remain the common attributes. The definition of isomorphic part in layout trees is generalized. According to the functionality [12], we roughly divide the nodes into four categories: "*clickable*", "*checkable*", "*editable*" and "*scrollable*". The nodes that belong to the same categories while with same depth in layout trees and same relative sequence in the layer are identified as isomorphic parts. We devise a greedy strategy to find the isomorphic parts. For the non-isomorphic parts, we apply a simply merge sort algorithm to adjust the order of nodes from two trees. Algorithm 1 illustrates the process of merging layout trees.

**Algorithm 1** Merge two layout trees to one matching tree

**Input:** Two layout trees *lt1* and *lt2*

**Output:** Minimum tree can match both *lt1* and *lt2*

```
1:  depth ← min(lt1.get_depth(), lt2.get_depth()) + 1
2:  root ← init_tree().get_root()
3:  lt1.get_root().set_root(root)
4:  lt2.get_root().set_root(root)
5:  for i = 0 → depth do
6:    let N_i = root.get_children_at_depth(i)        // matching
7:    for (v_a, v_b) ∈ greedy_match_isomorphic(N_i) do
8:      for each child ∈ v_b.get_children() do    //merging
9:        child.set_parent(v_a)
10:     end for
11:     v_a.get_parent().remove_child(v_b)
12:   end for
13: end for
14: return root
```

*D. Similarity Measurement*

We first save the app-level tree conducted by last operation into a document in XML format. And then we apply the CTPH [7] fuzzy hashing algorithm to measure the similarity between two apps. The CTPH algorithm generates a fuzzy hash fingerprint for every app by its *schema layout*. This is a one-time operation for each app as we stored this fingerprint in a SQLite database.

Before we make the pairwise comparisons among apps, a coarse-grained filter is leveraged to filter the app pairs that are not likely to be clones. The filter consists of three rules to filter out the suspicious app pairs:

- The clone app pairs must be signed with different signatures. The component reuse between the apps that released by the same developers is legitimate.
- The number of layout files left behind by *blacklist* in two apps should not be much different (i.e., the difference should be less than the smaller value of 1/5).
- The number of elements that under the same category (e.g., *clickable*, *checkable*, *scrollable*) in two apps should not be much different (i.e., the difference should be less than the smaller value of 1/5).

Different from traditional hash algorithms, the hash values of two similar files calculated by CTPH algorithm can still maintain the similarity. Hence, given two apps, we can easily calculate a similarity score for them by comparing the fuzzy hash fingerprints for their *schema layout*.

The original fingerprint comparison method in CTPH is to calculate the minimum edit distance (MED) between two fingerprints. We proposed WED (weighted edit distance) based on MED to measure the similarity between two fuzzy hash fingerprints. It is impossible to weight the elements by directly processing the hash fingerprint, because the fingerprint contains nothing about the hierarchical relationship of the elements. We proposed a prefix-based method to weight the elements in *schema layout* before generating the hash value, so that the original MED between two fingerprints also represents the weighted edit distance between them. Based on the calculated WED, we can derive a similarity score between two schema layouts. The formula is as follows:

$$SimilarityScore = \left[ 1 - \frac{WED}{\max\left(len_A, len_B\right)} \right] \quad (1)$$

The final similarity score is an integer in the range of zero to one hundred. Zero means two apps are completely independent and a higher score indicates that the app pair is more likely to be repackaged. If the similarity score of two apps is beyond the upper threshold $\theta$, we can identify these two apps are clone app. After evaluation, we found that when $\theta$ is set to 80, the accuracy (both false positive and false negative) of our approach hits the best.

## IV. EVALUATION

We conducted two sets of experiments to test the performance of SUIDroid. At first, we focus on the robustness of SUIDroid against the packed apps. Then, we evaluate the effectiveness and efficiency of SUIDroid on a large set of real-world apps. In addition, we measure the accuracy with a well-known clone app dataset provided by Chen et al. [17].

Our experiments are conducted on Linux with Dual-Core Pentium 2.50GHz CPU and 4GB memory.

### A. App Hardening Resilience

In order to comprehensively analyze the app hardening resilience of SUIDroid, we conduct two experiments on different packed app sets. In the first experiment, we evaluate the effectiveness of SUIDroid on six mainstream commercial Android packers, which cover the latest and most complex app hardening techniques. In the second experiment, we concern about the robustness on the real-world packed apps. We really appreciate the work Yang et al. [20] has done and the real world packed malware samples that they provided for us.

*1) Resilient to current mainstream commercial packers*

At first, we employ different commercial packers to encrypt the original app. Then, we use the SuiDroid to calculate the similarity pair wisely between the original app and the packed app. The higher similarity scores SuiDroid returns for each specific packer, the better resilience against that particular app harden service.

We randomly choose 20 apps from the Android app market based on different categories as the original app set and conduct a manual check to confirm that these 20 samples are all of different package names and contain enough layout files. We upload these test apps to the web portals of six mainstream packers (i.e., *Bangcle*, *Ijiami*, *Qihoo360*, *Tencent*, *Alibaba* and *Appfortify*) and finally get different packed versions. On this 140 Android app dataset, we evaluate our prototype together with two prior open source approaches. AndroGuard [3] is a *code-based* app clone detection tool. FSquaDRA [10] is an *UI-based* detection approach and is well known for its speed and resilient to code obfuscation techniques. Table 4 shows the app hardening resilience comparison between SUIDroid, AndroGuard and FSquaDRA. The *packer* columns indicate the names of harden service providers, while the *protection techniques* columns list the main packing techniques applied by these packers. The last three columns (SUIDroid, AndroGuard and FSquaDRA) list the average similar scores calculated by different approaches for each packer. Specifically, we separately apply the above three detection approaches to compute a similarity score for each original app and its packed version and finally report the average over 20 apps. Note that we normalized the original similar scores to make the comparison results more intuitive.

Overall, our experiment result turns out that SUIDroid is robust enough against current mainstream commercial packers. Comparing with previous detection approaches, no matter *code-based* or *UI-based*, our approach is much more effective and resilient to the app hardening techniques.

*2) Resilient to real-world packed samples*

It's necessary to evaluate our SUIDroid on real-world packed samples, because the hackers may employ some customized techniques or various hardening techniques together to evade the app clone detection in practice.

We test our approach with a dataset that contains 40 real-world packed malware samples. The dataset is accumulated from an online Android app analysis system, SandDroid [20] lasting for more than three years in collecting related packed malware samples. The experiment result tells that SUIDroid can effectively generate birthmark for most (39 out of 40) of the real-world packed malwares in dataset.

### B. Efficiency on Large-scale Real-World Dataset

We evaluate the efficiency of SUIDroid by a large scale of real-world apps. Specifically, we test the performance of each phase in SUIDroid during detecting the cross-market app clone.

*1) Dataset statics:* We crawled more than 100K (totally 104,831) Android apps from six app markets. The distribution of collected apps from different markets is shown in Table 2. Fig.3 shows the distribution of the sizes of APK files. Fig. 4 shows the average number of layout files in different size, also indicates the number of layout files that belong to third-party libraries in different sizes. There are nearly 60% (59.8K out of100K) of apps that include third-party libraries and 25.5 external layout files are included in each app on average. We randomly choose 15K apps to investigate the usage of third-party libraries in Android apps.

*2) Cross-market app clone detection performance:* We leverage the SuiDroid to detect the clone apps exist in different markets. SuiDroid required 155 hours to extract the layout XML files from all 100K app installation package files, or 5 seconds per app. The time cost in this phase will not be the bottleneck of our scalability, because this overhead for each app is just a one-time. It takes us another 89 hours to build the birthmark for each app, including filtering noisy-layout, merging layout trees and calculating fuzzy hash fingerprint. We build a SQLite database to store the birthmark and other information of each app. The database consumes less than 300MB of hard drive storage. The apps in the same pair are selected from different markets and the app pairs that released by the same developers or companies are removed. All app pairs (>109) that after the initial filtration are compared within 20 hours. Even compared with the

TABLE I. AVERAGE SIMILARITY SCORE CALCULATED BY SUIDROID COMPARED WITH ANDROGUARD AND FSQUADRA FOR EACH PACKER, '-' INDICATES NO SCORE RETURN

| Packer | SUIDroid | AndroGuard | FSquaDRA |
|---|---|---|---|
| Alibaba | 1.00 | - | 0.89 |
| Bangcle | 1.00 | - | 0.94 |
| Ijiami | 1.00 | 0.29 | 0.50 |
| Qihoo360 | 0.98 | - | 0.96 |
| Tencent | 0.98 | 0.17 | 0.75 |
| Appfortify | 1.00 | 0.65 | 0.56 |

TABLE II. REAL-WORLD EXPERIMENT DATASET

| Marketplace | Number of Apps | Size | Percentage |
|---|---|---|---|
| Anzhi | 25,829 | 129 GB | 25.51% |
| Appchina | 8,545 | 101 GB | 7.78% |
| Baidu | 6,411 | 49.3 GB | 5.83% |
| Gfan | 5,472 | 42.8 GB | 4.98% |
| Mumayi | 5,639 | 24.6 GB | 5.13% |
| Xiaomi | 57,967 | 457 GB | 50.77% |
| Total | 109,863 | 804 GB | 100% |

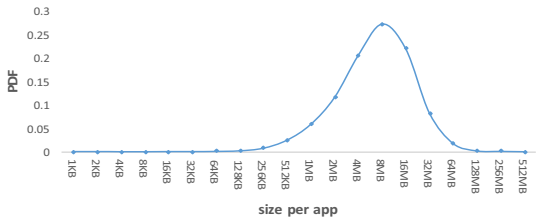


Fig. 3. The distribution of the size of APK files

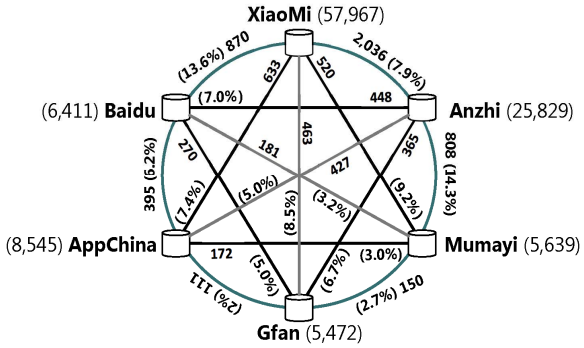


Fig. 5. The situation of cross-market app clones

FSquaDRA, which achieves an attractive processing speed like 6700 app pair comparisons per second, our approach still holds the advantage in terms of the efficiency of comparison algorithm. SuiDroid performs on average 10K app pair comparisons per second. Given an apk file, no matter packed or not, we can detect all the potential visual similar apps in our 100K apps dataset within on average 6 seconds. In particular, the pre-filter (at Section IV-D) can significantly reduce the number of unnecessary comparisons. Actually, almost 90% of the app pairs in dataset are eliminated directly.

**Determine the similarity threshold:** We need to determine a proper similarity threshold (Section IV-D) to try to reduce the inaccuracy of our approach. We use a series of similarity score thresholds to measure the accuracy. In practice, we randomly select 500 pairs of suspicious clone apps with various similarity scores, and manually determine the similarity of each app pair by checking the files in two apps, including layout files, manifest files and other multimedia files. Based on the false positives and the false negatives under different thresholds, we choose 80 as the optimal threshold. Fig. 6 shows the accuracy under different similarity score.

We present the situation of cross-market app clone in Fig. 5. Each node on outer circle corresponds to a market. The number and percentage besides an edge means the number of detected app clone pairs cross the two markets. Our detection result shows that the situation of cross-market app clone is still serious, as 2%-14% of apps in different markets are cloned.

## C. Accuracy

In this section, we use two sets of Android apps to evaluate the accuracy of SUIDroid. To measure the false negative rate (FNR) of our approach, we employ a well-known dataset [17] as the ground truth. In addition, we handpick 200 independent

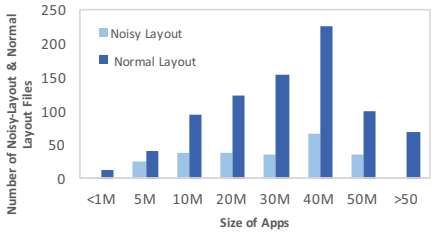


Fig. 4. The number of noisy and normal layout files in different size app

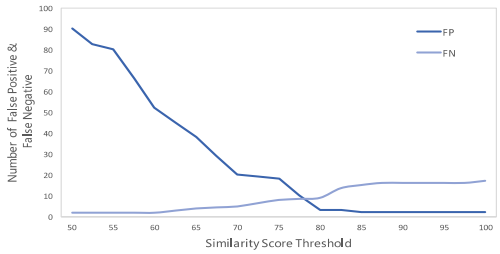


Fig. 6. The number of FN and FP in different similarity threshold

apps to build a dataset to verify the false positive rate (FPR) of SUIDroid.

*1) False Negative:* We consider the app pairs which are labeled as clones in the standard dataset while not detected by our approach to be false negative. The dataset includes 260 apps which are divided into 100 clone sets, with each group consisting of 2 or more apps. We manually check, install and compare these apps to make sure that all apps in the same group do have strong similarity. With the similarity threshold $\theta = 80$, we detected 98 sets of clones with 246 apps in total. Specifically, two sets of clones that detected by us are subsets of the same set in the original dataset. Therefore, with threshold $\theta = 80$, the false negative rate of SUIDroid is no more than 0.01%.

In these false negative cases, 4 sets of clones (11 apps) are different from other apps, as these apps are all mobile games based on Unity-3D engine and their user interface are defined in native source code. The remaining set contains 3 apps which run in background without any user interface. It is gratifying that after discarding these app pairs beyond our scope, our approach can achieve a 0.0% FNR.

*2) False Positive:* If a legitimate app is reported as clone app, we take it as a false positive. After analyzing many previous work [11, 12], we found out that the false positive occurs mainly under one case. When different apps were introduced into the same third-party libraries, the similarity between them may anomalously increase. Hence, we picked 200 apps of which 100 apps contain massive external libraries while the other half are with no third-party libraries as the test dataset to measure our false positive rate. These apps are independent of each other, except for the overlapping of the third-party libraries that they contained. We compared these

apps in a pairwise way and totally 19900 app pairs were completely compared.

When we set the similarity score threshold at 80, our approach detects only 12 false positive pairs. In other word, the false positive rate of SUIDroid is 0.06%. It is the *blacklist* that contribute to this low false positive rate. We selectly checked the birthmark of each app and found out that the *blacklist* can filter the *noisy-layout* in almost all apps.

## V. Related work

There are many recent studies focusing on detecting Android app clones. Most of the earlier work use sample code-based features.

DroidMOSS [18] adopts fuzzy hashing algorithm to calculate the similarity of the instruction sequence in two apps to detect clone apps. DNADroid [19] generates the program dependency graph (PDG) as a birthmark for each app and applies the sub-graph isomorphism algorithm to measure the similarity among PDGs. Chen et al. [17] leverages the methods information in apps to construct a 3D-control flow graph (3D-CFG). By comparing the cendroids of apps, they can detect the app clones quickly and accurately. Wukong [14] proposes a two-phase approach to detect app clones and use an accurate and automated clustering-based approach to filter external libraries.

Different from above code similarity based approaches, ViewDroid [8] proposes a user interface based approach which is resilient to code obfuscation techniques to detect app clones. FSquaDRA [10] detects app clones based on the comparison of the resource files in apps and is robust against obfuscation.

Charlie et al. [12] extracts the runtime UI information to construct birthmark for Android apps. SUIDroid is similar to their approach in the sense that both our approaches leverage on UI information. Our approach differs from theirs in that our birthmark features are extracted from static analysis and all these features are merged and organized in a tree structure.

## VI. Conclusion

In this paper, we proposed a novel *UI-based* Android app clone detection approach, SUIDroid. Specially, we only use the limit features extracted from layout XML files to build app birthmark. The evaluation results show that SUIDroid can effectively detect app clones among the packed apps, without compromising the legitimate app hardening service. We also proposed a counting based technique which can automatically and effectively filter the *noisy-layout* without prior knowledge for improving the accuracy of SUIDroid. Experiments on over 100K real-world apps show that SUIDroid maintains both accuracy and scalability at the same time.

## Acknowledgment

This research was supported in part by the National Natural Science Foundation of China (Project No. 61472125), and Research Foundation of Chinese Ministry of Education and China Mobile Communications Corporation (No. MCM 20122061).

## References

[1] IDC Corporate USA. Android Market Statistics from IDC. http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] VisionMobile. Mobile Application Economy Forcasts. http://www.visionmobile.com/product/app-economy-forecasts-2014-2017/

[3] A. Desnos and G. Gueguen. Android: From reversing to decompilation. In Black hat 2011, Abu Dhabi.

[4] AVL. Mobile Security Report for 2015. http://blog.avlyun.com/2016/02/2759/2015report/

[5] Apktool. https://code.google.com/p/android-apktool/, 2015.

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

[7] Kornblum J. Identifying almost identical files using context triggered piecewise hashing[J]. Digital Investigation the International Journal of Digital Forensics & Incident Response, 2006, 3(3):91-97.

[8] Zhang F, Huang H, Zhu S, et al. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection[C]//Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks. ACM, 2014: 25-36.

[9] Chen K, Wang P, Lee Y, et al. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale[C]//24th USENIX Security Symposium (USENIX Security 15). 2015: 659-674.

[10] Shao Y, Luo X, Qian C, et al. Towards a scalable resource-driven approach for detecting repackaged android applications[C]//Proceedings of the 30th Annual Computer Security Applications Conference. ACM, 2014: 56-65.

[11] Sun M, Li M, Lui J. DroidEagle: seamless detection of visually similar Android apps[C]//Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. ACM, 2015: 9.

[12] Soh C, Tan H B K, Arnatovich Y L, et al. Detecting Clones in Android Applications through Analyzing User Interfaces[C]// IEEE, International Conference on Program Comprehension. IEEE Press, 2015:163-173.

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

[14] Wang H, Guo Y, Ma Z, et al. WuKong: a scalable and accurate two-phase approach to Android app clone detection[C]//Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, 2015: 71-82.

[15] Zhu J, Wu Z, Guan Z, et al. Appearance similarity evaluation for Android applications[C]// Seventh International Conference on Advanced Computational Intelligence. IEEE, 2015.

[16] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: a scalable system for detecting code reuse among android applications. In Proc. DIMVA, 2012.

[17] Chen K, Liu P, Zhang Y. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets[C]// International Conference on Software Engineering. ACM, 2014:175-186.

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

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

[20] Yang W, Zhang Y, Li J, et al. AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware[M]//Research in Attacks, Intrusions, and Defenses. Springer International Publishing, 2015: 359-381.

[21] Zhang Y, Luo X, Yin H. Dexhunter: toward extracting hidden code from packed android applications[M]//Computer Security--ESORICS 2015. Springer International Publishing, 2015: 293-311.