Hindawi
Mobile Information Systems
Volume 2017, Article ID 6958698, 12 pages
https://doi.org/10.1155/2017/6958698

Research Article

An Efficient and Packing-Resilient Two-Phase Android Cloned Application Detection Approach

Fang Lyu, Yaping Lin, and Junfeng Yang

College of Computer Science and Electronic Engineering, Hunan University, Changsha, China

Correspondence should be addressed to Yaping Lin; yplin@hnu.edu.cn

Received 24 March 2017; Accepted 23 August 2017; Published 10 October 2017

Academic Editor: Salvatore Carta

The huge benefit of mobile application industry has attracted a large number of developers and attendant attackers. Application repackaging provides help for the distribution of most Android malware. It is a serious threat to the entire Android ecosystem, as it not only compromises the security and privacy of the app users but also plunders app developers' income. Although massive approaches have been proposed to address this issue, plagiarists try to fight back through packing their malicious code with the help of commercial packers. Previous works either do not consider the packing issue or rely on time-consuming computations, which are not scalable for large-scale real-world scenario. In this paper, we propose FUIDroid, a novel two-phase app clones detection system that can detect the packed cloned app. FUIDroid includes a function-based fast selection phase to quickly select suspicious apps by analyzing apps' description and a further UI-based accurate detection phase to refine the detection result. We evaluate our system on two sets of apps. The result from experiment on 320 packed samples demonstrates that FUIDroid is resilient to packed apps. The evaluation on more than 150,000 real-world apps shows the efficiency of FUIDroid in large-scale scenario.

1. Introduction

As Android grows into the most popular mobile OS by global market share, Android-targeted attacks continue rising in both number and complexity [1]. More than 85% of Android malware rely on *application repackaging* to spread to a large number of genuine users [2]. In this paper, we define repackaged apps (a.k.a., cloned apps) as applications that impersonate a genuine app, by slight modifications to the genuine one (e.g., crack paid apps to bypass payment function, modify the advertisement libraries, or even insert various malicious functions) in a way that the cloned app still looks and feels like the original one. By "copying" the user interface (UI for short) of a genuine app, cloned app can easily target novice users that gravitate towards the popularity of the genuine application.

App repackaging now is becoming one of the most serious threats to the whole Android ecosystem, which seriously violates the intellectual property and profit of the legal developers. Moreover, the cloned app that carries malicious code may even reveal users' privacy and threaten their property security.

To address this imminent issue, previous studies have proposed various approaches to detect the repackaged apps. These approaches can be divided into two categories: *code-based* techniques [3–8] and *UI-based* techniques [9–17]. Code-based methods focus on detecting the code reuse in repackaged apps, while UI-based methods analyze the similarity of app's user interface.

It is encouraging that several app clone detection systems [5, 12] have claimed to achieve good performance in real-world scenario. However, the commercial app packing service, which is intended for protecting genuine apps from being disassembled and repackaged at first, is now becoming an effective practice for illegal apps to thwart current detection systems. It is disturbing that so far at least 10% of Android malware is maliciously packed [18] and, unfortunately, we find that all code-based methods and most UI-based methods lack the resiliency to the commercial packing techniques, which can encrypt app's source code (i.e., the *.dex* file) dynamically [19].

Although few latest UI-based approaches can detect the packed apps, they are not scalable in real-world scenario, because all of these approaches heavily depend on either complex computation based on app's resource files [13, 17], or app's run-time information [14, 15] which is difficult to be extracted and time-consuming.

In this paper, we propose FUIDroid, a novel two-phase cloned app detection system that is resilient to the packed app and achieves accuracy and scalability simultaneously. FUIDroid includes a function-based fast selection phase to quickly select all suspected clones of the target app by comparing their function descriptions, and a UI-based accurate detection phase to perform detailed UI-based comparison between the previously selected candidate clone pairs to refine the detection result.

FUIDroid is motivated by the following observations: first, to deceive users, a cloned app must provide the same core functionality just as the genuine one does, together with a "looks/feels like" user interface. Second, the features about app's functionality and user interface are stored separately from the source code, which means they are free from current app packing techniques [19]. Lastly, inspired by WuKong [4], multilevel based detection strategy is able to keep great advantage in large-scale scenario.

Specifically, in the fast selection phase, we apply some mature nature language processing techniques [20, 21] to analyze the similarity of app's functional description. In the detection phase, we propose an UI-based birthmark called *schema layout* to identify apps. Schema layout is a compact XML document that retains the specifics of all layout files in the app. We apply a state-of-the-art tree-based edit distance algorithm [22, 23] to calculate the similarity between birthmarks to determine cloned apps.

FUIDroid is expected to achieve scalability and accuracy at the same time, because our balanced binary tree-based searching scheme in fast selection phase can effectively narrow down the number of suspicious cloned apps by several orders of magnitude and the well-designed UI-based birthmark we construct in the detection phase can improve the final accuracy of the entire detection system. Since all the features (functions and UI) extracted by FUIDroid can keep constant after app packing service, our system is resilient to current commercial packing techniques. Our evaluation results demonstrate our points.

In summary, our paper makes the following contributions:

(i) We propose FUIDroid, a novel two-phase Android cloned app detection system that is able to detect the packed apps while achieving accuracy and scalability in real-world scenario.

(ii) In the fast selection phase, we improve a fast searching scheme based on multikeyword tree index to select potential cloned apps, which is much more efficient than traditional pairwise comparison methods.

(iii) In the detection phase, we propose a new UI-based birthmark, called schema layout, for Android app. The highlight is that it only relies on the limited features extracted from the layout files, while still achieving great accuracy.

(iv) We implement a prototype system and evaluate the packing resilience of FUIDroid on 40 real-world packed app samples collected from SandDroid [24]. The experimental result shows that FUIDroid is able to handle the app packed by current mainstream commercial app packers.

(v) We evaluate our approach on over 150,000 real-world apps crawled from 6 popular Android markets. It turns out that 3%–14% of apps are cloned cases. We also evaluate the accuracy of FUIDroid on a set of cloned apps detected by a well-known detection tool [25]. The false positive rate is only 0.06%.

The rest of this paper is organized as follows. Section 2 introduces two essential pieces of background knowledge related to our work and claims the scope of our paper. The design goals and the overview structure are presented in Section 3. Sections 4 and 5 focus on the core ideas and the implementation details of the selection and detection phase. Evaluation is presented in Section 6, followed by related work in Section 7. Finally, we conclude the work in Section 8.

2. Background

In this section, we will introduce some essential background about the structure of Android app and the commercial app packing service. At last, we give the scope of our paper.

2.1. Android App Structure. Android app is distributed and installed in Android app package (i.e., **APK**) format. An APK is an archive that contains the *.dex* file and other resource files. The *.dex* file is a Dalvik executable file compiled from Java source code and implements the functionality of app. Due to the openness of Java programing language, it is easy to reverse engineer the *.dex* file with the help of some open source tools [26–28] to extract app's source code. All the raw resource files, including images, audios, and the XML files that define the layout of user interface, are stored in separate folders (like **/res/drawable** or **/res/layout**). The information about app developers is stored under the **MATE-INF** folder.

Besides the files conducted by developers, there are various external files (icons, Java file, layout file, etc.) imported by third-party libraries in most of APK files. These files mix with original files and may influence the accuracy and efficiency of app clone detection.

2.2. Android Packing Service. App packing service is intended to protect Android app from being reversed, modified, and repackaged [18, 19]. Commercial security companies apply various code protection techniques to hide the *.dex* file and impede the attempt of dumping the source code. *Code obfuscation* [29] is used to raise the bar of understanding the logic of source code. *Dynamic loading* helps packers to encrypt the original *.dex* file to prevent it from being disassembled and decrypt it before running the app. *Antidebugging* can detect the running environment and thwart the debugging *gdb* by self-attaching. All these techniques are designed to protect the *.dex* file, because all the source code is assembled in it.

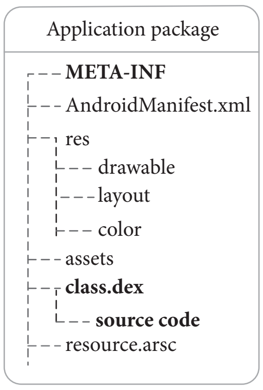


Figure 1: The structure of APK and the packed parts are marked by boldface.

Figure 1 shows the structure of normal APK and the boldfaced words represent the files encrypted by packing service.

App packing is becoming the common practice for single app to resist malicious cracking. Unfortunately, the app packing service is available for all apps without strict security analysis. More and more malware tries to use packing techniques to disguise the malicious functions to evade the security detection in app markets. Although there are some approaches [30, 31] that can unpack these packed apps, they all rely on the a priori knowledge obtained by manual analysis.

2.3. Scope and Assumptions. In this paper, our purpose is to detect cloned Android app pairs but not to identify which is the original one and which is the cloned one.

2.3.1. Inconsistent Functional Description. The description of an app on the third-party market may be missing or not match the apps implementation. Although the consistent description is not essential in our detection system, it can greatly improve our detection speed. Hence, apps with inconsistent description will increase the time complexity of our method. Our paper does not concern these apps, because (a) this kind of app arouses the vigilance of users very easily and (b) the inspection by the app market is becoming more and more strict.

2.3.2. Dynamic User Interface. We only focus on the apps whose user interface 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. Just like previous static detection systems, apps whose user interface is dynamically defined by programs are also out of the scope of our paper. All of the apps used in our evaluation are not paid apps, so we can crawl enough apps to simulate the real-world scenario.

3. System Design

3.1. Design Goals. Our system is proposed to help app market to detect Android app clones in real world. Considering the current realistic detection scenario, there are three key goals for FUIDroid.

(i) Packing Resilience. There are more and more malware employee commercial packing services to encrypt or obscure their source code to evade detection [18]. Hence, FUIDroid should be able to handle the packed apps.

(ii) Efficiency and Scalability. The number of apps in real world has reached millions and is increasing every day [32]. Therefore, our detection approach must be scalable to detect apps in large-scale.

(iii) Accuracy. Accuracy is a basic goal for detection system. The key is to construct an accurate birthmark to characterize app.

3.2. Overview of FUIDroid. Figure 2 shows the architecture and workflow of FUIDroid. The whole system contains two consecutive phases: a function-based fast selection phase and an UI-based accurate detection phase.

In the selection phase, we apply natural language processing techniques to extract keywords from app's functional description to construct a feature vector. Suspicious cloned apps are quickly selected by the tree-based multikeyword searching algorithm [33]. In the detection phase, we extract layout trees [13] from apps layout files to build an UI-based birthmark to calculate similarity scores for app pairs selected by the coarse-grained phase.

4. The Function-Based Fast Selection Phase

4.1. Challenge and Strategy. In our design, the task of the selection phase is to quickly pick up a small set of candidate cloned apps for the target app in large-scale scenario. By dramatically narrowing down the number of suspicious app pairs, the selection phase can reduce the invalid computation in next accurate detection phase and ultimately improve the scalability of the whole system. The performance of searching algorithm and the false alarm rate together determine the performance of selection scheme. Therefore, the key challenge for our selection scheme is: *how to achieve scalability in large-scale scenario with low false alarm rate?* We figure out that there are two realistic challenges that hinder our function-based selection scheme from achieving scalability and low false alarm rate.

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

Challenge 2 (C2). It is difficult to get low false positive rate and false negative rate at the same time, especially when we take *C1* into consideration. We try to reach our goals in two ways:

(i) To overcome *C1*, for each single app, we only extract the function-based features from app's description to avoid any time-consuming operations. For apps under the same category, we modify an efficient tree-based searching algorithm [33] to quickly select the

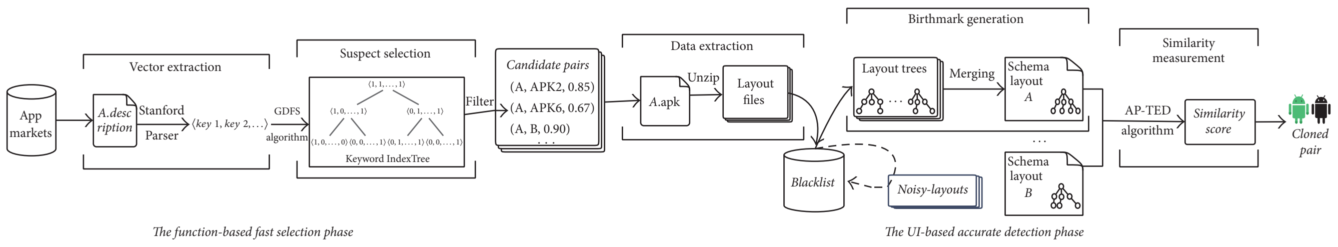


Figure 2: The architecture and workflow of FUIDroid.

Table 1: Examples of the workflow of vector extraction.

App	Part of description	Keywords	Vector
Uber	...There's no need to park or wait for a taxi or bus. With Uber, you just tap to request a ride, and it's easy to pay with credit or cash in select cities...	{...taxi, need, uber, bus, city, ride, credit, cash...}	⟨...4, 26, 38, 7, 40, 10, 869, 0...⟩
Twitter	...From breaking news and entertainment, sports and politics, to big events and everyday interests. Be part of what everyone is talking about and get videos, live footage and Moments, direct from the source...	{...news, entertainment, sport, politics, event, interest, video, footage, moment, source...}	⟨...27, 39, 25, 1021, 269, 1347, 225, 9926, 484...⟩
Tinder	...is a powerful tool to meet people, expand your social group, meet locals when you're traveling and find people you otherwise never would have met...	{...tool, people, group, locals...}	⟨...30, 246, 270, 331...⟩

suspicious cloned apps to reduce redundant pairwise comparisons.

(ii) To overcome *C2*, we apply the Stanford Parser [21] to preprocess the description before extracting key words to reduce the false negative rate. To reduce the false positive rate, we set up a simple filter to filter out the obvious independent app pairs.

4.2. Implementation. We implement the fast selection system with 600 lines of Java code to parse original description and extract key words vector and 1,000 lines of Python code to build keyword index tree and perform large-scale searching task. For every target app, the complete process includes two steps: extracting features from app's description and searching suspicious cloned apps by comparing features with target app.

4.2.1. Feature Extraction. We execute a crawler program to download apps and descriptions from third-party market. After analyzing 40,000 app description samples, we select 15,000 most used words to construct a standard dictionary, which covers most of the functionality of the apps. Different from previous methods [34–38], which all directly use app's description, for single input app, we try to construct a function-based vector based on app's description. The original description is usually very detailed and redundant; hence we apply Stanford Parser to work out the grammatical structure of each sentence in description and filter out the dispensable parts like empty words in description. Finally, only the function related words are left. All left words are transformed to original form by [20] to avoid deviations caused by tenses or plurals (i.e., plays/played to play). After that, we construct a dynamic-dimensional vector for the target app and fill the vector with a set of integer values according to the index of the extracted keywords in the standard dictionary. Table 1 gives examples of the workflow of feature extraction.

4.2.2. Suspect Selection. Regular distance-based similarity measurement method for vector, like cosine distance, is not suitable for our large-scale scenario because of the complex computing. In [33], Xia et al. proposed a tree-based multikeyword searching approach that has claimed to achieve sublinear search efficiency. We make modifications to this system and apply it to quickly search similar apps. Specifically, all the keyword vectors are stored with the structure of keyword balanced binary (KBB) index tree [39, 40] and the construction details are described in [33]. We make three major improvements to the original algorithm. First, we construct a specially designed dictionary for app functional description, which contains 15,000 keywords. Second, to avoid complex computation, we transform the fixed high-dimensions vector to dynamic-dimensions vector to store features. Third, we skip all computations related to privacy and security. The searching process is a recursive procedure upon the index tree and finally all suspicious cloned apps whose relevance scores with the target app beyond an upper threshold are returned. Apps that share high relevance scores are divided into the same group and wait for further detection in the fine-grained phase.

4.2.3. Filtering Strategy. Before the complex pairwise comparisons, a filter is applied to filter out the obvious independent app pairs incorrectly selected by the fast selection system. We consider the fact that under two conditions related apps should not be grouped together:

(i) The suspicious app pairs signed with the same signature. The component reuse among apps released by the same developers is common.

(ii) The sizes of installation file of two apps should not be much different (i.e., the difference should be less than the smaller value of 1/3).

5. The UI-Based Accurate Detection Phase

5.1. Challenge and Strategy. The major goal of UI-based detection system is to accurately determine the cloned app pairs in a small set of suspicious apps that are selected by previous fast selection system. To improve the performance of FUIDroid in real-world scenario, our UI-based detection system should meet strict requirements on both accuracy and packing-resiliency. Through analysis, we found that current mainstream app packing (i.e., app hardening) services and app developers only pay attention to the security of the source code (i.e., the *.dex* file) in app, while leaving the layout files nonencrypted. Thus, our detection approach tries to avoid conflicts with current packing services by proposing a new birthmark that only relies on the UI-based features extracted from layout XML files to identify apps. In this case, here is the main challenge in the accurate detection system that we need to overcome: *how to build an accurate birthmark with limited resources?* We construct a new UI-based birthmark, called *schema layout*, which uses the limited features extracted from layout XML files to accurately characterize apps. Below is the formal definition of *schema layout*.

Definition 1. A schema layout is a prototype XML document subsuming the most relevant layout features of all the layout files within a single app, and the following applies:

(i) *The elements and attributes in schema layout are extracted from the layout files through static analysis.*

(ii) *The relative relationships (e.g., including, containing, and sequence) among elements in original layout files are retained in schema layout.*

(iii) *The homogeneous subparts in different layout files are merged in schema layout, while the noisy-layout is not involved in the generation of schema layout.*

Schema layout is distinct enough to identify the Android apps and the detailed analysis and construction process are left to later in this section.

The key obstacle that hinders *schema layout* from achieving a higher accuracy is the *noisy-layout*. The *noisy-layout* refers to two particular types of layout files: **(NL-1)** the external layout files included by the third-party libraries and **(NL-2)** the extra layout files imported by plagiarists. We proposed a counting based filtration technique, called *blacklist*, to filter the *noisy-layout*. *Blacklist* identifies the *noisy-layout* by analyzing the frequency of the layout file in a great number of apps:

(i) Within a large enough app database, the layout files that belong to *NL-1* are those of which the frequency exceeds an upper threshold, because third-party libraries would be included by many different apps.

(ii) Inside the clone app couple (app and its clone version), the native layout files are copied once, while the *NL-2* files only exist in the repackaged app. Thus, the local frequency of *NL-2* inside the couple is half of the native layout files.

5.2. Implementation. The accurate detection task can be divided into three steps. For each suspicious app selected by the UI-based selection system, we first extract the layout features from app's installation file (i.e., the **APK**) and then use the *blacklist* to filter out *noisy-layout*. After filtration, we use these features to construct a unique UI-based birthmark and apply the TED (Tree-based Edit Distance) [41] to measure the similarity between birthmarks.

5.2.1. Feature Extraction. We extract the layout features from app to build birthmark to identify Android app. The UI-based birthmark is reliable and effective because

(i) cloned apps share similar layout files to keep the look and feel similar to original apps to deceive users and it takes the plagiarist great efforts to reimplement an existing layout from scratch;

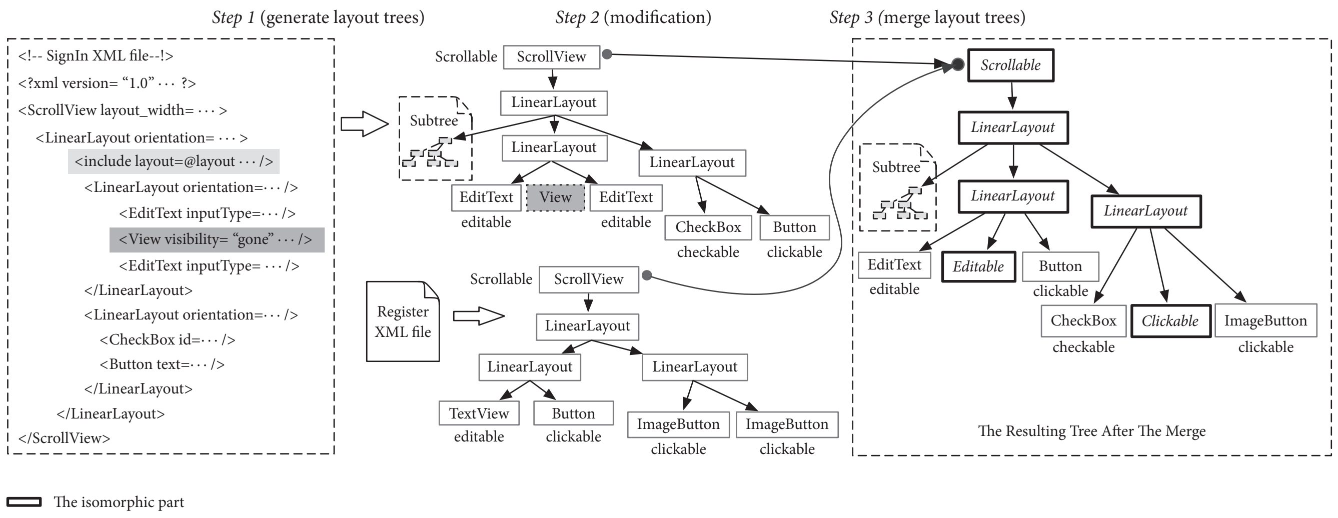
(ii) the layout files are stored separately in APK and stay consistent after app packing service;

(iii) it is a lightweight operation to extract layout features as we only obtain the layout files from APK, rather than disassembling the entire app.

A major drawback of our approach is that the layout-based birthmark can be easily tainted by *noisy-layout*. We proposed a counting based filtering method, called *blacklist*, to address this issue. The layout feature extraction can be broken down into two steps.

Obtain Layout Files. The user interface of Android app is usually defined in layout XML files. For each app, we obtain layout files from the */res/layout* directory. Some apps are carefully packaged to resist reverse engineering. In this case, we apply the "*unzip*" command together with *AXMLPrinter* (a format conversion tool) to extract layout files.

Filter Noisy-Layout. The extracted layout files can be classified into 2 groups: internal layout files created by app developers and external layout files *(noisy-layout)* imported by third-party 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 SQLite database which stores the MD5 hash fingerprint of all identified *noisy-layout* and can update automatically. To find out the external layout files in the app,



```
<!-- SignIn XML file--!>
<?xml version= "1.0" ··· ?>
<ScrollView layout_width= ··· >
  <LinearLayout orientation= ··· >
      <include layout=@layout ··· />
        <LinearLayout orientation=··· />
            <EditText inputType=··· />
            <View visibility= "gone" ··· />
            <EditText inputType= ··· />
        </LinearLayout>
        <LinearLayout orientation=··· />
          <CheckBox id=··· />
          <Button text=··· />
        </LinearLayout>
    </LinearLayout>
</ScrollView>
```

Figure 3: An example of generating birthmark.

we compare the hash value of layout file with the hash values stored in the database. If they match, we update the frequency of this file and identify the *noisy-layout* only if its frequency exceeds the upper threshold. For example, we find out that the layout file *alipay.xml* exists in more than 700 independent apps (among 15 K apps). We can determine that this layout is a typical *noisy-layout* that belongs to a well-known third-party library (i.e., AlipaySDK).

5.2.2. Birthmark Generation. Software birthmark is a unique characteristic of a certain app that can be used to determine the identity of app. For Android app, birthmark can be divided into two categories: code-based birthmark and UI-based birthmark. In this phase, we proposed an UI-based birthmark, *schema layout*, that relies on app's layout features for app clones detection. Figure 3 illustrates the process of constructing our UI-based birthmark. The construction consists of three steps.

Generate Layout Trees. The layout features of apps are separately defined in layout XML files. To simplify next merge operation, we generate a layout tree [13] 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 (button, image, text, etc.) in user interface, while the nonleaf 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.

Modify Layout Trees. We need to modify the original layout trees to filter the useless elements and attributes that may cause ambiguity between the layout file and the user interface. There are some elements in layout files that make no difference to the actual user interface but may affect the accuracy of our birthmark. Most 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. Thus, we need to clear the attributes and remove the nodes with zero width and height or whose visibility is set to "invisible" in layout tree.

Merge Layout Trees. After building layout trees over separate layout files, we merge these trees together to generate a final matching tree for the entire app. Considering the accuracy and efficiency, we need to reduce the number of redundant nodes as far as possible while maintain the diversity of layout files. In summary, the output of this step is a minimal tree that contains the structure of all separate layout trees. 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 their direct children. The definition of isomorphic part in layout tree is generalized. According to the functionality [14], we divide the element into four categories: "clickable," " *checkable*," " *editable*," and "scrollable." Nodes that belong to the same categories while with the same depth in one layout tree are identified as isomorphic parts. We devise a greedy breadth-first strategy to find the isomorphic parts. Algorithm 1 illustrates the process of merging layout trees.

5.2.3. Similarity Measurement. We save the tree conducted by previous algorithm to database in the bracket notation. Then we apply **AP-TED** (All Path Tree Edit Distance) [22] algorithm to measure the similarity between two apps. AP-TED algorithm measures similarity of tree structured data by using the TED measure. The TED between ordered labeled trees is the minimal-cost sequence of node edit operations (*delete*, *insert*, or *replace*) that transforms one tree into another. Comparing to other TED algorithms, AP-TED algorithm is more efficient as it consumes less memory for the strategy computation than for the actual distance computation, which is the bottleneck of previous algorithms.

```
    Input: Two layout trees $lt_1$, $lt_2$;
    Output: Minimum tree match both $lt_1$ and $lt_2$;
(1) $depth = \min(lt_1.\text{getDepth}(), lt_2.\text{getDepth}()) + 1$;
(2) $root$ = initTree ().getRoot ();
(3) $lt_1$.getRoot ().setRoot (root);
(4) $lt_2$.getRoot ().setRoot (root);
(5) for $i = 0$ to $depth$ do
(6)     $N_i$ = $root$.getChildrenAtDepth ($i$);
(7)     foreach ($V_a$, $V_b$) in MatchIsomorphic ($N_i$) do
(8)         foreach child in $V_b$.getChildren () do
(9)             child.setParent ($V_a$);
(10)        end
(11)        $V_a$.getParent ().removeChild ($V_b$);
(12)    end
(13) end
(14) return $root$;
```

Algorithm 1: Merge two layout trees.

After computation, we normalized the AP-TED output to get the final similarity score. The mapping formula is as follows:

$$\text{Similarity Score} = \left[1 - \frac{\text{TED}}{\max(\text{len}_A, \text{len}_B)}\right] \times 100. \quad (1)$$

The final similarity score is an integer of zero to a hundred. Zero means two apps are completely independent and a higher score indicates that the app pair is more likely to be cloned. If the similarity score of two apps is beyond the upper threshold θ, we can identify that these two apps are cloned app. After evaluation, we found that when θ is set to 90, the accuracy (both false positive and false negative) of our approach is optimal.

6. Evaluation

We conducted two sets of experiments to evaluate the performance of FUIDroid. At first, we focus on the robustness of our UI-based detection approach against the packed apps. Then, we evaluate the effectiveness and efficiency of the whole system on a large set of real-world apps. In addition, we measure the accuracy with a set of cloned apps detected by AndroGuard (a well-known Android app diagnostic tool).

6.1. App Packing Resilience. It is necessary to declare at first that only the fine-grained detection phase is tested, because the selection phase does not directly handle the packed APK. In order to comprehensively analyze the app packing resilience of FUIDroid, we conduct two experiments on different packed app sets. In the first experiment, we evaluate the effectiveness of FUIDroid on six mainstream commercial Android packers, equipped with the latest packing techniques. In the second experiment, we are concerned about the robustness in the real-world packed apps. We really appreciate Yang et al. [18] for providing us with the real-world packed malware samples.

Table 2: Average similarity score calculated by FUIDroid compared with AndroGuard and FSquaDRA for each packer; "—" indicates no score return.

Packer	AndroGuard	FSquaDRA	FUIDroid
Alibaba	—	**0.98**	0.94
Bangcle	—	0.94	**1.00**
Ijiami	0.29	0.90	**1.00**
Qihoo360	—	**1.00**	0.96
Tencent	0.17	0.85	**0.98**
Tongfudun	—	0.96	**0.99**

6.1.1. Resilience to Current Mainstream Commercial Packers. At first, we employ different commercial packers to encrypt the original app. Then, we use the UI-based detection approach in FUIDroid to calculate the similarity between the original app and the packed app. The higher similarity scores returned by FUIDroid for each specific packer indicate the better resilience against that particular app packing service.

We randomly choose 40 apps from Android app market as the original app set and conduct a manual check to confirm that these 40 samples are all of different package names and contain enough layout files. We upload these apps to the web portals of six mainstream commercial packers (i.e., *Bangcle* [42], *Ijiami* [43], *Qihoo360* [44], *Tencent* [45], *Alibaba* [46], and *Tongfudun* [47]) and finally get different packed versions. On this 280-Android app dataset, we evaluate our system together with two well-known open source tools. AndroGuard is a code-based app clones detection tool. FSquaDRA is an UI-based detection system and is well-known for its detection speed and is resilient to code obfuscation. Table 2 shows the app packing resilience comparison among FUIDroid, AndroGuard, and FSquaDRA. The packer columns indicate the packing service providers. The next three columns (FUIDroid, AndroGuard, and FSquaDRA) list the average similarity score calculated by different systems for six packers. Specifically, we apply the above three detection systems to measure the similarity for each original app and its packed version and finally report the average similarity over 40 apps. Note that we normalized the similarity scores to make the comparison results more intuitive.

Overall, our experiment result show that FUIDroid is robust enough against current mainstream commercial packers.

6.1.2. Resilient to Real-World Packed Samples. It is essential to evaluate FUIDroid on real-world packed samples, because the hackers may employ some customized techniques or hybrid packing techniques to evade the detection system 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, lasting for more than four years in collecting related packed malware samples. We manually disassemble every malware sample to make sure that all these samples are packed. The experiment result shows that FUIDroid can effectively generate birthmark for most (39 out of 40) real-world packed

Table 3: Real-world experiment dataset.

Market	Number of apps	Size	Percentage
Anzhi	49,855	255 GB	32.06%
Gfan	5,472	42.8 GB	3.51%
Google	7,939	47.6 GB	5.11%
HiAPK	13,199	156 GB	8.49%
SnapPea	11,055	105 GB	7.11%
Xiaomi	67,967	536 GB	43.71%
Total	155,487	1,397 GB	100%

Table 4: The top 6 most used noisy-layout and libraries.

Noisy-layout	Library	Frequency
abc_list_menu_∗.xml	Android/support/v7	3186/15520
abs_list_menu_∗.xml	ActionBarSherlock	2268/15520
umeng_socialize∗.xml	UmengSDK	2236/15520
pull_to_refresh_∗.xml	PullToRefresh	2086/15520
slidingmenumain.xml	SlidingMenu	951/15520
alipay.xml	AlipaySDK	715/15520

malware samples. The only failed case is specially encrypted and cannot be disassembled by common reverse engineering tools or simple "*unzip*" command.

6.2. Efficiency on Large-Scale Real-World Dataset. We apply FUIDroid to detect the cloned apps that exist in different markets to evaluate the efficiency of each subsystem in our system in real-world large-scale scenario. Furthermore, we analyze the time consumption of FUIDroid in the experiment.

6.2.1. Dataset Statics. We crawled more than 150 K (totally 155,487) Android apps and descriptions from six app markets. The distribution of collected apps from different markets is shown in Table 3. We list the top 6 most used noisy-layout and libraries in 15 K app samples in Table 4. Figure 4 shows the average number of original words and final extracted words in app descriptions that are under different categories. Figure 5 shows the average number of layout files in app with different size and also indicates the number of layout files that belong to third-party libraries in different size. There are nearly 65% (97.4 K out of 150 K) of apps that include third-party libraries and 26 external layout files are included in each app on average.

6.2.2. Function-Based Fast Selection Phase

Feature Extraction. The dependency parser we used is able to parse more than 1000 sentences or about 100 app descriptions per second. We construct a dynamic-dimensions vector for each app and filled in integer values based on the index of the key words that we extracted within 1 ms. As shown in Figure 4, more than 90% apps contain more than 140 words in their description and, after parsing, only 24 key words are finally distilled on average.

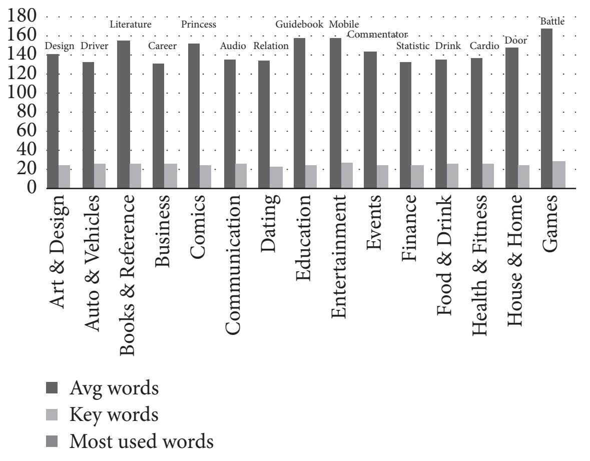


Figure 4: The number of words and keywords extracted from app descriptions under different category.

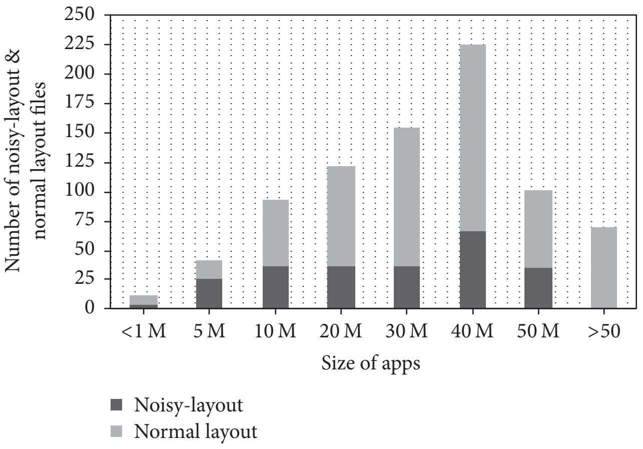


Figure 5: The number of noisy and normal layout files in different size apps.

Suspect Selection. According to the category provided by the app market, we construct 15 binary index trees for 150 K apps with an average depth of 13. With nonparallel execution, our selection scheme can accomplish the task of searching suspects for 1000 target apps per second. In most of positive cases, the selected suspicious apps only accounted for 0.025% of the original category. Due to the special structure of index tree, the complexity of searching scheme is fundamentally kept logarithmic.

Determine Threshold. The accuracy of searching scheme is directly affected by the relevance threshold (see Section 4.2.2). Hence we choose 30,000 samples (2,000 apps per category) and apply a series of relevance thresholds to measure their accuracy. After manually checking sampling, we find that, with a threshold of 0.85, our selection scheme can achieve the most true positives. Figure 6 shows the selection result with different relevance score.

Although there exist some false positive cases, we can reduce the final false positive rate in next UI-based detection phase. With the determined threshold, we detect all the apps

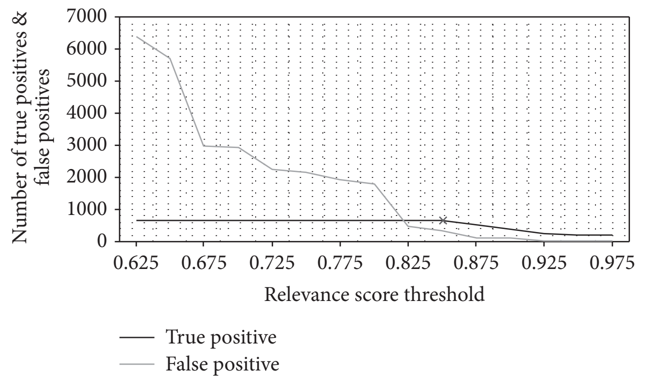


Figure 6: The number of true positives and false positives under different relevance score. The sign "×" refers to the final threshold used in our system.

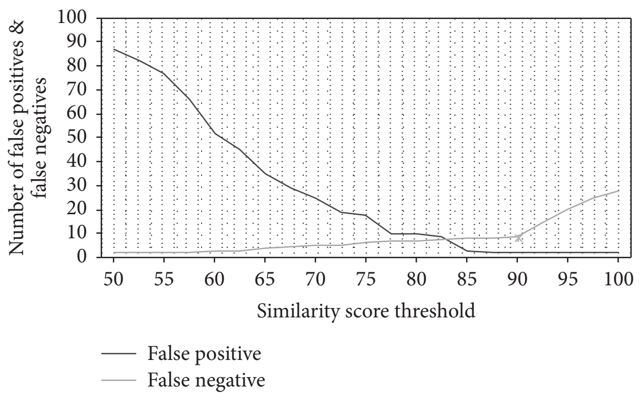


Figure 7: The number of false positives and false negatives under different similarity score. The sign "×" refers to the final threshold used in our system.

in our database. We get 1.5×10^4 distinct suspicious cloned app pairs and on average 0.2 cloned suspect for each target app. After selection, the size of suspicious cloned apps to each app is greatly reduced. Ultimately, the selection phase narrows down the number of UI-based pairwise comparison by almost five orders of magnitude (from 10^4 narrowed down to 10^{-1}) to enhance the scalability of the entire system.

6.2.3. UI-Based Accurate Detection Phase

Feature Extraction. FUIDroid requires 166 hours to extract the layout XML files from 150 K app installation package files, or 4 seconds per app. The time cost in this phase will not be the bottleneck of our scalability, because this overhead for each app happens one time.

Birthmark Generation. It takes us another three hours to build the birthmark for each app, including filtering noisy-layout, merging layout trees, and saving to database. We build a SQLite database to store the birthmark and other information of each app. The database consumes less than 400 MB of hard drive storage.

Similarity Measurement. All suspicious cloned app pairs (3×10^4) are detected within one hour. Even compared with FSquaDRA, which is well-known for the attractive processing speed like 6,700 pairwise comparisons per second, our approach still holds the advantage in terms of the efficiency of detection algorithm. Although our UI-based detection method performs only 43.5 pairwise comparisons per second, the previous coarse-grained selection phase already helps us avoid massive invalid comparisons. Given an APK and its description, no matter packed or not, we can find out the potential cloned apps in our 150 K apps dataset within on average 5 seconds.

Determine Threshold. We apply a series of similarity thresholds (see Section 5.2.3) to measure the accuracy of our approach. In practice, we randomly select 500 pairs of suspicious clone apps with different similarity scores and manually identify the cloned app pairs by checking the files in two apps, including layout files, manifest files, and other multimedia files. By counting the false positives and the false negatives under different threshold, we choose 90 as the optimal threshold. Figure 7 shows the accuracy under different similarity score.

We present the situation of cross-market app clones in Figure 8. Each node on outer circle corresponds to a market. The number and percentage besides an edge means the number of app clone pairs cross the two markets. Our detection result shows that the situation of cross-market app clone is still serious, as 3%–14% of apps in app markets are cloned.

6.2.4. Time Consumption.

We evaluate the time consumption of both FUIDroid and FSquaDRA using the same collected app dataset on a MacBook Pro laptop with 2.9 GHz Intel i7 processor with 2 cores, and 8 GB 1867 MHz DDR3 memory. FSquaDRA is another famous app clone detection system, which is well known for its very low time complexity. Table 5 shows the detail time consumption of FUIDroid and FSquaDRA on 10,000 apps. During the detection phase, FUIDroid can only finish about 43.5 pairwise similarity comparisons per second, which obviously has disadvantages to FSquaDRA. While in the preprocessing phase, besides the common operations like unzip and reverse engineering, FUIDroid conducts extra function-based selection operation, which can greatly reduce the number of the target apps pairwise comparisons. The extra phase does not cause extra time overhead because it is executed in parallel with other preprocesses. Hence, the preselection operation can effectively reduce the time consumption of FUIDroid's entire detection process.

In summary, given a target app, FUIDroid can find out all the cloned versions from 10,000 apps within 5 seconds.

6.3. Accuracy.

In this section, we use two sets of Android apps to evaluate the accuracy of FUIDroid. To measure the false negative rate (FNR) of our approach, we employ a small set of clone apps detected by AndroGuard (a well-known open source detection tool) as the ground truth. In addition, we

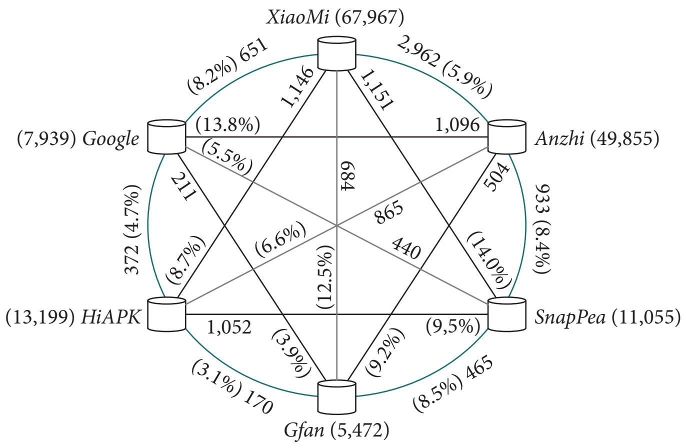


Figure 8: The situation of cross-market app clones.

Table 5: The average time consumption of detecting one cloned application in dataset.

Approach	Preprocesses	Time of preprocess	Time of detection	Average competition times	Total time consumption
FSquaDRA	unzip, reverse engineering	4 sec	0.15 ms	10000/10000	5.5 sec
FUIDroid	suspect selection, unzip, reverse engineering	4 sec	23.0 ms	18/10000	4.4 sec

handpick 200 independent apps to build a dataset to verify the false positive rate (FPR) of FUIDroid.

6.3.1. 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 AndroGuard dataset includes 92 apps which belong to 67 clone pairs (25 pairs, ten 3-app sets and three 4-app sets). We manually check, install, and compare these apps to make sure that all apps in the same set do have strong similarity. When the relevance threshold is set to 0.85, the function-based selection system obtains 117 distinct suspicious cloned app pairs that cover all the 67 positive cases. In the UI-based detection system, with similarity threshold = 90, we finally determined 66 cloned app pairs. Hence the overall false negative rate of FUIDroid is no more than 1.49%.

After further investigation, these two apps in the false negative case are different from normal apps, as they are developed based on *Unity-3D* engine and their user interface is defined in native source code. It is gratifying that, discarding the apps beyond our scope, FUIDroid can achieve a perfect FNR.

6.3.2. False Positive. If a legitimate app is reported as cloned app, we take it as a false positive. We consider that the false positive rate of FUIDroid is actually determined by the performance of UI-based detection system. After analyzing many previous works [4, 8–10], we find out that the false positive occurs mainly under one case. When different apps include 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 third-party libraries that they contained. We compared these apps in a pairwise way and totally 19,900 app pairs are completely compared.

When we set the similarity threshold at 90, our approach detects only 12 false positive pairs. In other words, the false positive rate of FUIDroid is 0.06%. It is the *blacklist* that contributes to this low false positive rate. We checked sampling of the birthmark of each app and found out that the *blacklist* can filter almost all *noisy-layout* in apps.

7. Related Work

There are many studies focusing on detecting Android app clones. Most of the earlier works use sample code-based features to build birthmark to identify app.

DroidMOSS [6] adopts fuzzy hashing algorithm to calculate the similarity of the instruction sequence in two apps to detect app clones. DNADroid [7] generates the program dependency graph (PDG) as a birthmark for each app and applies the subgraph isomorphism algorithm to measure the similarity between PDGs. Chen et al. [8] use the calling sequence of API in app's source code to construct a 3D-control flow graph (3D-CFG). By comparing the birthmark of apps, they can detect app clones quickly and accurately. Parmjit and Sharma [35] propose a hybrid approach to analyze application. This hybrid approach depends upon three parameters—Description Mapping, Interface Analysis, and Source Code Analysis—to identify a unique application. Wukong [4] proposes a two-phase approach to detect app clones and uses an accurate and automated clustering-based approach to filter third-party libraries.

One of the major drawbacks of code-based systems is that their detection accuracy is easily affected by code obfuscation. To address this issue, recent studies focus on using UI features to identify apps.

ViewDroid [10] constructs a novel birthmark based on the switch sequence of UI which is resilient to code obfuscation techniques to detect app clones. FSquaDRA [17] detects app clones by comparing the resource files in app which is fast but not robust against modifications to the resource files. Soh et al. [14] and Malisa et al. [15] try to execute app on a specific simulator to extract run-time UI information to construct birthmark for Android apps, which is robust to malicious modifications but the time-consuming execution ruined their scalability in large-scale scenario. FUIDroid is similar to their approaches in the sense that they all rely on UI information. Our approach is more advanced as our UI-based birthmark is extracted from static analysis and, by working with the function-based selection, massive invalid comparisons are avoided.

8. Conclusion

In this paper, we proposed a novel two-phase Android app clone detection system, FUIDroid. We only use, especially, the limited features extracted from app description and layout XML files to quickly and accurately identify the cloned apps. The evaluation results show that FUIDroid can effectively detect app clones among the packed apps, without compromising the legitimate app packing service. We also proposed a counting based filtering method to improve the accuracy of FUIDroid which can effectively filter the *noisy-layout* without any prior knowledge. Experiments on over 150 K real-world apps show that FUIDroid achieves both accuracy and scalability at the same time in large-scale scenario.

Disclosure

This paper is an extended version of the conference paper presented in the 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM2016) [48].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

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," Website, 2017. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, Calif, USA, May 2012.

[3] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy, CODASPY 2013*, pp. 185–195, February 2013.

[4] H. Wang, Y. Guo, Z. Ma, and X. Chen, "WuKong: A scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 24th International Symposium on Software Testing and Analysis, ISSTA 2015*, pp. 71–82, July 2015.

[5] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: a scalable system for detecting code reuse among android applications," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 62–81, 2012.

[6] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the the Second ACM conference*, pp. 317–326, San Antonio, Texas, USA, Feburary 2012.

[7] J. Crussell, C. Gibler, and H. Chen, *Attack of the Clones: Detecting Cloned Applications on Android Markets*, Springer, Berlin, Germany, 2012.

[8] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pp. 175–186, June 2014.

[9] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014*, pp. 56–65, December 2014.

[10] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2014*, pp. 25–36, July 2014.

[11] J. Zhu, Z. Wu, Z. Guan, and Z. Chen, "Appearance similarity evaluation for Android applications," in *Proceedings of the 7th International Conference on Advanced Computational Intelligence, ICACI 2015*, pp. 323–328, March 2015.

[12] K. Chen, P. Wang, L. Yeonjoon et al., "Finding unknown malice in 10 seconds: mass vetting for new threats at the google-play scale," in *Usenix Security Symposium*, pp. 659–674, 2015.

[13] M. Sun, M. Li, and J. C. S. Lui, "DroidEagle: seamless detection of visually similar android apps," in *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2015*, pp. 1–12, June 2015.

[14] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, "Detecting clones in android applications through analyzing user interfaces," in *Proceedings of the 23rd IEEE International Conference on Program Comprehension, ICPC 2015*, pp. 163–173, May 2015.

[15] L. Malisa, K. Kostiainen, M. Och, and S. Capkun, "Mobile application impersonation detection using dynamic user interface extraction," *Lecture Notes in Computer Science (including*

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9878, pp. 217–237, 2016.

[16] I. Gurulian, K. Markantonakis, L. Cavalaro, and K. Mayes, "You can't touch this: Consumer-centric android application repackaging detection," *Future Generation Computer Systems*, vol. 65, pp. 1–9, 2016.

[17] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "Fsquadra: Fast detection of repackaged applications," in *Proceedings of the The Ifip Wg 11.3 Working Conference on Data and Applications Security and Privacy*, pp. 130–145, 2014.

[18] W. Yang, Y. Zhang, J. Li et al., "AppSpear: Bytecode decrypting and DEX reassembling for packed android malware," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9404, pp. 359–381, 2015.

[19] Y. Zhang, X. Luo, and H. Yin, *DexHunter: Toward Extracting Hidden Code from Packed Android Applications*, Springer International Publishing, 2015.

[20] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller, "Introduction to wordnet: an on-line lexical database," *International Journal of Lexicography*, vol. 3, no. 4, pp. 235–244, 1990.

[21] D. Chen and C. D. Manning, "A fast and accurate dependency parser using neural networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*, pp. 740–750, October 2014.

[22] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," *ACM Transactions on Database Systems*, vol. 40, no. 1, pp. 1–40, 2015.

[23] M. Pawlik and N. Augsten, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157–173, 2016.

[24] Botnet Research Team from Xi'an Jiaotong University. Sanddroid - an automatic android application analysis system. Website, 2017. http://sanddroid.xjtu.edu.cn/.

[25] Geoffroy Gueguen Anthony Desnos. Androguard. Website, 2017. https://github.com/androguard/androguard.

[26] Ryszard Winiewski Connor Tumbleson. Apktool. Website, 2017. https://ibotpeaches.github.io/Apktool/.

[27] JesusFreke. Smali/baksmali. Website, 2017. https://github.com/JesusFreke/smali.

[28] Google. Dex2jar. Website, 2017. https://github.com/pxb1988/dex2jar.

[29] GuardSquare nv. Proguard - java code obfuscation. Website, 2017. https://www.guardsquare.com/en/proguard.

[30] Y. Haoyang, *Towards Unpacking Android Apps [PhD. Thesis]*, Department of Computing, The Hong Kong Polytechnic University, 2016.

[31] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 358–369, May 2017.

[32] Google. Wiki - google play. Website, 2017. https://en.wikipedia.org/wiki/Google_Play.

[33] Z. Xia, X. Wang, X. Sun, and Q. Wang, "A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data," *IEEE Transactions on Parallel & Distributed Systems*, vol. 27, no. 2, pp. 340–352, 2016.

[34] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the description-to-permission fidelity in Android applications," in *Proceedings of the 21st ACM Conference on Computer and Communications Security, CCS 2014*, pp. 1354–1365, November 2014.

[35] K. Parmjit and S. Sharma, "Spyware detection in android using hybridization of description analysis, permission mapping and interface analysis," *Procedia Computer Science*, vol. 46, pp. 794–803, 2015.

[36] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*, pp. 538–549, July 2016.

[37] L. Yu, X. Luo, C. Qian, and S. Wang, "Revisiting the description-to-behavior fidelity in android applications," in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 415–426, March 2016.

[38] L. Yu, X. Luo, C. Qian, S. Wang, and H. K.N. Leung, "Enhancing the description-to-behavior fidelity in android apps with privacy policy," *IEEE Transactions on Software Engineering*, vol. 99, pp. 1-1, 2017.

[39] B. Gu and V. S. Sheng, "Feasibility and finite convergence analysis for accurate on-line v-support vector learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 24, no. 8, pp. 1304–1315, 2013.

[40] X. Wen, L. Shao, W. Fang, and Y. Xue, "Efficient feature selection and classification for vehicle detection," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 3, pp. 508–517, 2015.

[41] K. C. Tai, "The tree-to-tree correction problem," *Journal of the Association for Computing Machinery*, vol. 26, no. 3, pp. 422–433, 1979.

[42] Bangcle. Bangcle. Website, 2017. https://dev.bangcle.com/.

[43] Ijiami. Ijiami. Website, 2017. http://www.ijiami.cn/.

[44] 360.CN. Qihoo360. Website, 2017. http://jiagu.360.cn/.

[45] Tencent legu. Tencent legu. Website, 2017. http://legu.qcloud.com/.

[46] Alibaba. Alibaba. Website, 2017. http://jaq.alibaba.com/.

[47] Tongfudun. Tongfudun. Website, 2017. https://www.tongfudun.com/protect.jhtml.

[48] F. Lyu, Y. Lin, J. Yang, and J. Zhou, "SUIDroid: An efficient hardening-resilient approach to android app clone detection," in *Proceedings of the Joint 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 10th IEEE International Conference on Big Data Science and Engineering and 14th IEEE International Symposium on Parallel and Distributed Processing with Applications, IEEE TrustCom/BigDataSE/ISPA 2016*, pp. 511–518, August 2016.