A GUI Based Dynamic Birthmark Generation Method for Android Applications

Shixuan Zhao,[1,*] Ming Wang,[1] and Zihan Zeng[1]

[1] *Department of Computer Science and Technology, Nanjing University*

(Dated: April 2, 2019)

Abstract: The fast growing of modern smart devices, especially smartphones, has dramatically expanded the industry of software. However, along with the fast-growing industry, malicious software created by application repackaging using reverse engineering tools has become one of the most severe threats to the security and privacy of users as well as the right of developers. A good way to fight against application repackaging is to use software birthmark, which extracts certain unique features and characteristics from the software. Hence we proposed and implemented a GUI based dynamic birthmark Interface-Style-Icon Tree or ISI tree, and its generation method which uses the icon of each user interface as the birthmark to overcome the flaws of current birthmark generation methods. Our method employs a DFS with limited depth to travel through each of the user interface and collects and analyses icons then build a tree with these icons to create the final birthmark. We also use machine learning to recognize the style of each icon to reduce the workload of the final comparison.

I. INTRODUCTION

The fast growing of modern smart devices, especially smartphones, has dramatically expanded the industry of software. However, along with the fast-growing industry, malicious software created by application repackaging using reverse engineering tools[1][2][3] has become one of the most severe threats to the security and privacy of users as well as the right of developers.[4]

A good way to fight against application repackaging is to use software birthmark[5], which extracts certain unique features and characteristics from the software, satisfying the following two properties:[5]

1. Resistance: If program p' is obtained from p by any program semantic preserving transformation, the birthmarks of both p' and p should be the same. Obfuscation and encryption, shouldn't affect the birthmarks of the transformed programs.

2. Credibility: For program p and q implementing the same specifications, if p and q are written independently, the birthmarks of p and q should be different.

Birthmark can be generated statically or dynamically. Static birthmarks, like [6]-[12], are usually credible, however, may not resist code obfuscations[13] and could even hardly work when it comes to app encryption.[14] Currently, dynamic birthmarks technologies like [15] which takes advantage from API informations and [16] which collects runtime UI informations are naturally resistant to code obfuscations. But by inserting useless but transparent activities or controls, these methods can easily waste a lot of time on useless informations which would heavily impact the performance.

However by combining dynamic birthmark technologies with GUI informations, we can create a very effective birthmark generation mathod. Using GUI informations is intuitive since malware creators want their repackaged software to look as close as the original one thus victim consumers would trust that the repackaged one is the same as the original one. One of the key features of a GUI is its icons. To create two undistinguishable software, one needs their user interface looks really similar thus requires a set of icons that is as close to the original as possible. This leads to a method of dynamically collecting icons from each interface and use them as the birthmark of an app.

In this paper, we proposed and implemented a GUI based dynamic birthmark Interface-Style-Icon Tree or ISI tree, and its generation method which uses the icon of each user interface as the birthmark to overcome the flaws of current birthmark generation methods. Our method employs a DFS with limited depth to travel through each of the user interface and collects and analyses icons then build a tree with these icons to create the final birthmark. We also use machine learning to recognize the style of each icon to reduce the workload of the final comparison. We implemented it and compared it with current approaches.

II. METHODOLOGY

A. Interface-Style-Icon Tree

To represent the final birthmark, we define a structure named *Interface-Style-Icon Tree* or ISI tree. An ISI tree is a directed graph with each vertex represent a user interface and a collection of its icons and the styles of icons. We now formally define the ISI tree.

Definition 1. ISI tree: An ISI tree is a directed graph $G = (V, E)$, whose V is the set of vertices representing each interface and E is the set of edges representing the transition between interfaces. □

Definition 2. Interface vertex: An interface vertex is a tuple $v = (I, S, l) \in V$, whose I is the set of icons of current interface, S is a mapping of $I \to \mathbb{N}$ that maps

* shixuan.zhao@hotmail.com

icons to a style numbered with natural numbers and l is the depth of the interface during the DFS procedure. □

Definition 3. Edge: An edge $e \in E \subseteq V \times V$ represent a transition from one to another interface during the DFS procedure. □

B. Similarity Calculation

Similarity calculation between two ISI trees is intuitive. We only compare vertices on the same depth since the same interface should most likely to be triggered on the same depth of the full UI transition tree of the app. After calculating each depth's similarity, we sum them with their weights to create the final score of similarity.

We start with the method of comparing two interfaces. For any two interfaces v_1, v_2 having the same l value, We first build a bipartite graph G_b.

Definition 5. $G_b = (V_b, E_b)$, in which

$$V_b = v_1.I \cup v_2.I \tag{1}$$

and

$$E_b = \{(i_1, i_2) | i_1 \in v_1.I, i_2 \in v_2.I, v_1.S(i_1) = v_2.S(i_2)\}. \tag{2}$$

□

That is, G_b is a bipartite graph whose vertices are icons from v_1 and v_2 and icons are connected only if they have the same style value. We then derive another bipartite graph $G'_b = (V_b, E'_b)$ from G_b with the only difference is that an edge in G'_b should not only in G_b, but also has the connected two icons be the *same* (using methods like pHash[17]), which, in a formal way, is

Definiton 6. $G'_b = (V_b, E'_b)$, in which

$$E'_b = \{(i_1, i_2) | (i_1, i_2) \in E_b, same(i_1, i_2)\}. \tag{3}$$

□

By building G'_b, we are actually doing a maximum matching problem, with the cardinality of the match be the number of same icons. G_b right here reduces the overhead by preventing comparison between two icons that are different in style which would result in a total waste of time to compare them in a graphical way. The similarity score between two interface v_1, v_2 can now be formally defined.

Definiton 7. Interface similarity score:

$$\varphi(v_1, v_2) = \frac{|\text{MAX-MATCH}(G'_b)|}{|v_1.I| + |v_2.I|} \tag{4}$$

□

We now define the similarity between vertices from two ISI trees G_1, G_2 in the same depth. We are still employing maximum matching method to solve this problem. For each depth l, a bipartite graph $G_{bl} = (V_{bl}, E_{bl})$ is built whose V_{bl} contains two copies of the vertices in depth l from each of the two ISI trees and E_{bl} contains the edge between interfaces that are considered same. A formal definition is

Definition 8. $G_{bl} = (V_{bl}, E_{bl})$, in which

$$V_{1l} = \{v_{1l} | v_{1l} \in V_1, v_{1l}.l = l\} \tag{5}$$

$$V_{2l} = \{v_{2l} | v_{2l} \in V_2, v_{2l}.l = l\} \tag{6}$$

$$V_{bl} = V_{1l} \cup V_{2l} \tag{7}$$

and

$$E_{bl} = \{(v_1, v_2) | v_1 \in V_{1l}, v_2 \in V_{2l}, \varphi(v_1, v_2) > \delta_\varphi\}. \tag{8}$$

□

Here, δ_φ is a preset value that indicates the threshold exceeding which would lead two interfaces be considered similar. Thus the similarity between vertices in depth l can be expressed as

$$\frac{|\text{MAX-MATCH}(G_{bl})|}{|V_{bl}|} \tag{9}$$

We then sum all the similarity scores with weights and then normalize it, which leads to the final score of similarity between two ISI trees:

Definition 9. ISI tree similarity:

$$\varphi(G_1, G_2) = \frac{\sum_{l=1}^{\delta_l} |\text{MAX-MATCHING}(G_{bl})|}{\sum_{l=1}^{\delta_l} |V_{bl}|} \tag{10}$$

□

C. ISI Tree Generation

The generation of an ISI tree is basically about running the app, collecting icons and analyse the style, generating interactions (like press a button) to transit to another interface.

The whole framework is a DFS, with each item in the stack is an interface. Due to our limited depth, we can use an implicit stack created by calling function DFS recursively.

Algorithm 1 describes the ISI tree generation procedure. We start by running DFS on the initial interface (line 5). In our DFS procedure, we dump

the interface's icons and analyse each icon's style, then combined with its depth, we can create a three tuple node of the ISI tree which exactly represents current interface (line 12-20). After building up the current tuple, we add it to the ISI tree's vertices. Then we choose δ_c randomly picked interactable controls then interact with them, and it would take as to a some new interfaces i' which we would recursively do DFS on it and add an edge from current interface to the new interface in our ISI tree (line 23-28). Note that δ_c here is a preset value which limits the breadth of our search while δ_l limits the depth. The DFS procedure will return a reference to the vertex it created thus we can add edge to our ISI tree (line 27).

It might be hard to actually return from a new interface because we do not even know if there is a return button, thus we only save the controls and their activities instead of trying to return. After a DFS procedure, we can perform transition to the next interface by triggering controls' activities.

Algorithm 1 ISI Tree Generation

```
1: let G = (V, E) be the ISI tree with empty V, E
2: procedure ISI TREE GENERATION
3:     let l = 0 be an integer
4:     let i_0 be the initial interface
5:     let root = DFS(i_0, 0)
6:     return root
7: end procedure
8:
9: procedure DFS(i, l)
10:    let cnt = 0 be a counter
11:    let I, S, C be empty sets
12:    let v = (I, S, l) be a new tuple
13:    C = DUMP-CONTROLS(i)
14:    for all c ∈ C do
15:        if c is an icon then
16:            add c into I
17:            add (c, GET-STYLE(c)) into S
18:        end if
19:    end for
20:    add v to G.V
21:    if l < δ_l then
22:        while cnt < δ_cnt do
23:            randomly choose a control c from C
24:            if INTERACTABLE(c) then
25:                interact with c to a new interface i'
26:                let v' = DFS(i', l + 1)
27:                add (v, v') to G.E
28:                cnt+ = 1
29:            end if
30:        end while
31:    end if
32:    return reference of v
33: end procedure
```

III. IMPLEMENTATION

A. Overview

We implemented the method with three modules. The DFS strategy module which is the core execution manager and the ISI tree generator, is implemented in Java. The style classifier, which classifies icons into different categories, is implemented in python and is called by the DFS strategy module. The final module is the similarity comparison module, which calculates the final similarity score between two interfaces.

Our implementation can run on Linux and Windows. We are current targeting Android 5.0 and does not care running on a VM or on a physical device.

B. ISI Tree in Practice

The formal structure of ISI tree needs some transformation to be easier to implement. We implemented ISI trees by adding attributes to icons. In fact, we use the filenames of the dumped icons to store these attributes. The file name is constructed as

`APPID_INTERFACEID_STYLE_ICONID.png`

Thus we are actually doing a mapping

$$I \to V \times \{S\}. \tag{11}$$

Because when we are doing similarity calculating we are actually manipulating with icons (the set I), this allows as to quickly grab information from the set I instead of querying from a actual tree.

C. Running an App with DFS

We implemented our birthmark generator using UI Automator, which is a popular testing framework for Android apps.[19] The process starts by installing the target app using `adb`. When the app is installed, we launch the app with debugger attached, then start up the UI Automator.

For each interface, we dump the controls into a list then save icons from the list with a reduced file name as

`APPID_INTERFACEID_ICONID.png`

and launch the style analyzer to get the style number, append it into file name in the form described in the previous subsection.

Now we have finished processing the current interface, we check our depth l. If $l < \delta_l$, we pick a random control from the list and interact with it. After interacting with it, we consider ourselves in a new interface. Remember that we are doing DFS with an implicit stack by calling it recursively, thus the list of controls is actually located in the stack frame of the current call of DFS.

D. Dumping the Icons and Controls

Dumping controls can be tricky. One of the hardest problem is to filter out useless transparent controls added by the hacker who repackaged the app. We here propose a handy filter named UIValidator. The UIValidator accepts a control's position and a screenshot, calculates the entropy of the area and the edges of the control, then returns whether the control is valid. For an image which is focused the entropy is usually large, whereas the entropy of an unfocused image is usually smaller.[18] Thus we would expect a valid control's edge's entropy should be small, but the central area's entropy should be large. And by predefining two threshold δ_{el} and δ_{eu}, we can have the UIValidator algorithm described in **Algorithm 2**.

By validating each of the controls and decide whether it is an icon, we now have to extract the icon and save it to a file. The challenge here is that there are possible solutions to load an icon without being captured by UI Automator, thus instead of intercepting with loading event, we use the position information to clip the icon off the screenshot.

Algorithm 2 UIValidator

Input: control c and screenshot s
Output: a bool value indicates if the control is valid

```
procedure UIValidator(c, s)
    let e_e and e_c be the edge and center entropy
    e_e = entropy(c.edge, s)
    e_c = entropy(c.area, s)
    if e_e < δ_el ∧ e_c > δ_eu then
        return true
    else
        return false
    end if
end procedure
```

E. Style Recognition

Style recognition is considered a method to accelerate the performance of the similarity calculation process. Doing image similarity calculation is resource-demanding and we want to reduce the number of images that needs to be compared. By dividing icons into different type of styles can significantly decrease the number.

In our implementation, we employed a machine learning method to classify icons into 7 categories. These categories are defined by colors, skeuomorphism or not, etc. We trained our model with TensorFlow [20] and achieved a fast classification process. The model is pretrained and is ready to use out of the box. The classifier was written in python and the DFS procedure described before will call a python interpretor to execute the classifier and get the returned result.

F. Similarity Calculation

The similarity is calculated as described in the previous section. Considering that an altered version of icons an be hard to tell by human but a total difference to the computer, like slightly change the color of certain pixels, or adding a transparent line of pixels, naive image matching is not a very good method.

To overcome this issue, we calculate the similarity between icons using pHash,[17] which calculates the perceptual hash between two images and is robust to these modification.

IV. EVALUATION

A. Dataset

We used the same dataset from a previous work in our research group, the RepDroid.[21]

Our dataset S is made by randomly picking apps from Wandojia,[22], one of the most popular Android app market in China, and mixed it with 3 duplicateed but encrypted apps to check our performance.

To build S, we simply picked the top-downloaded apps from Wandojia, and each of them has been downloaded at least 100,000 times. S has a size of 101 apps.

B. Comparison

We have applied both of these two data sets to the ViewDroid,[12] Soh et al.'s work[16](called UI Method), and Kim et al.'s workref:DAVA(called API Method).

Note that only 30 apps of S can run on Android 2.2.3 used by *API Method*.

The comparison result is in TABLE I.

TABLE I. Performance on Dataset

	S			
	False Positive	False Negative	FP Rate	FN Rate
Ours	0	0	0%	0%
ViewDroid	108	2	2.10%	33.33%
UI Method	191	0	3.71%	0%
API Method	6	1	1.38%	16.67%

We get a really satisfying result since most of the encryption methods will not try to mess up the real user interface, thus our approach would hardly fail.

C. Threats to Validity

We have made the best effort to ensure the validity of the dataset but there are yet some threats to the validity of the result.

The UI Method's source code is not available, thus we used the implementation from our research group which could undermine our result. The API Method is too old to run on the latest version of Android which makes newer apps which refused to run on older version of Android remained out of test. And the dataset only consists of free apps which might not be well generalized.

V. LIMITATION AND FUTURE WORKS

Our implementation using UI Automator is not yet hacked but there are indeed methods to fully dysfunctionalize UI Automator. A workaround is to transfer to the Hierarchy Viewer,[23], which uses reflecting method rather than uses the XML that could be easily cheated.

There also exists a kind of apps that our work cannot deal with which are the web apps. Using HTML5 to build software has become a trend in recent years. Frameworks like PhoneGap[24] are not using native controls but web controls thus leads to a really difficult task to analyse them.

Another important issue that deserves a good discussion is how human tell the difference between two interfaces. We now consider that human can tell a difference if the icons change a lot. But to define "a lot", we still need help from psychologists.

Our method only takes icons as the key characteristic of an interface, while there are plenty of other features including color distributions that can be put into practice.

VI. CONCLUSION

We designed a GUI based birthmark generation method which models the GUI behaviours of Android apps to detect application repackaging. Our approach dynamically executes the target app to build the ISI tree which represents the birthmark and use it to evaluate the similarity between apps.

[1] Apktool. A tool for reverse engineering android apk files. [Online].
Available: https://ibotpeaches.github.io/Apktool/

[2] dex2jar. Tools to work with android .dex and java .class files. [Online].
Available: https://sourceforge.net/projects/dex2jar/

[3] Soot. A framework for analyzing and transforming java and android applications. [Online].
Available: https://sable.github.io/soot/

[4] M. Ballano, "Android threats getting steamy," [Online] February, vol. 28, 2011.

[5] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs." in IASTED Conf. on Software Engineering, 2004, pp. 569–574.

[6] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in Proceedings of the second ACM conference on Data and Application Security and Privacy. ACM, 2012, pp. 317–326.

[7] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in European Symposium on Research in Computer Security. Springer, 2012, pp. 37–54.

[8] J. Crussell, C. Gibler, and H. Chen,"Scalable semantics-based detection of similar android applications," in Proc. of Esorics, vol. 13. Citeseer, 2013.

[9] 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 International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2012, pp. 62–81.

[10] 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. ACM, 2013, pp. 185–196.

[11] 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. ACM, 2014, pp. 175–186.

[12] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: to- wards obfuscation-resilient mobile application repackaging detection," in Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks. ACM, 2014, pp. 25–36.

[13] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in 2012 IEEE Symposium on Security and Privacy. IEEE, 2012, pp. 95–109.

[14] M. Dalla Preda and F. Maggi, "Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology," Journal of Computer Virology and Hacking Techniques, pp. 1–24, 2016.

[15] D. Kim, A. Gokhale, V. Ganapathy, and A. Srivastava, "Detecting plagiarized mobile apps using api birthmarks," Automated Software Engineering, pp. 1–28, 2015.

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

[17] pHash: The open source perceptual hash library. [Online].
Available: phash.org

[18] Ch. Thum (1984) Measurement of the Entropy of an Image with Application to Image Focusing, Optica Acta: International Journal of Optics, 31:2, 203-211, DOI: 10.1080/713821475

[19] UI Automator | Android Developers. [Online].
Available: https://developer.android.com/training/testing/ui-automator

[20] TensorFlow. [Online].
Available: https://www.tensorflow.org

[21] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu, "RepDroid: An Automated Tool for Android Application Repackaging Detection", 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 132-142, 2017.

[22] Wnadoujia. [Online].
Available: http://www.wandoujia.com/apps

[23] Profile your layout with Hierarchy Viewer | Android Developers. [Online].
Available: https://developer.android.com/studio/profile/hierarchy-viewer

[24] PhoneGap. [Online].
Available: https://phonegap.com