A Similar Module Extraction Approach for Android Malware
Zheng-qiang Li\(^1,2\), Yan-chen QIAO\(^1,*\), Touhidul Hasan\(^1,2\) and Qing-shan JIANG\(^1\)
\(^1\)Shenzhen Institutes of Advanced Technology, Shenzhen, China
\(^2\)University of Chinese Academy of Sciences, Beijing, China
*Corresponding author

Keywords: API sequences, Similar module, Android malware, Key function call graph.

Abstract. Android is the popular mobile operating system, and it has been attracting many developers and malware software authors into the field. It is becoming critical to identify the malicious program in the large count of mobile applications, whereas similarity comparison methods have been proposed earlier to detect malware. However, most of the works focus on detecting malicious program from benign and malware, and they did not consider the details of similarity between malicious programs. In this paper, we propose an approach based on key function call graph to extract similar module between malware which could be used to detect malicious programs on Android platform. The proposed method employs Android system API call sequences to construct the similar module between two malicious programs. The experiments on real-world dataset demonstrate that the proposed approach is effective for extracting similar modules between malware.

Introduction
As of October 2017, according to StateCounter [1], Android mobile operating system market share is 73%. The vast market share of Android attract legitimate developers and malware authors to engage in this area for significant profits. According to the McAfee threat report, new Android malware rose by 60% in the third quarter of 2017 [2]. The rise of malicious programs is creating the challenge to protect the users from infectious Android applications. Malware creators do not write a new malware from the scratch since this will take too many efforts to complete the job. Conversely, they use automated tools such as repacking, hooking, and injector to develop malware quickly from the legitimate applications and distribute them in different Android markets. These will induce multiple duplicate code segments among many malware [3]. As a result, many studies based on similarity comparison have been proposed to meet the challenge. Most of the works have samples containing both benign and malware applications. And, the published works did not consider the similarity between malware.

In this paper, we propose an approach of extracting similar modules of malware from API sequences on the Android platform. Malware are classified using the results of VirsuTotal [4], and all the malware are decompiled into smali codes using Apktool [5]. We construct the function call graph from smali code and build the key function call graph based on the function call graph. Finally, similar modules between malware are extracted. The experimental results and analysis show that the proposed approach is effective to extract similar modules.

Background and Related Work
In this section, we review the methodologies [6-10] for Android malware detections and their limitations. Furthermore, we discuss key function call graph for the proposed approach.

Android Malware Detection Methodologies
to classify both applications and games. Peiravian and Zhu [8] classify applications with privileges and API calls, and they establish a classifier that can be used to identify whether a program is a malicious program.

These published works were focused on the detection or classification of malware. They did not consider the details of the similarities between malware software. The similarities between modules were considered in code-based work to malware detection. Qiao and Yun [9] proposed a method based on simhash and inverted index to detect function reuse from high-volume code. In the code-based work, the similar code blocks are extracted and determine whether the applications are similar based on the calling relationship between function codes. Ruttenberg [10] proposed a method to extract similar modules. The complexity of code similarity determination is high, resulting in less efficient and unable to adapt to the rapid growth of malware. Unlike the above studies, our method is based on the Android system call sequence and can be used to extract similar modules between malware effectively.

**Key Function Call Graph**

Any user function that calls two or more Android system APIs is called a *key function*. When a function is not a *key function*, then we call it a *non-key function*. In this paper, Android system APIs refers to the function provided by the Java API framework layer [11].

The *key function call graph (KFCG)* consists of key functions [12]. Function call graph is used to represent the calling relationship between function blocks. Let KFCG = (V, E), where V and E represent the vertices and edges of the graph KFCG, respectively. KFCG is a directed acyclic graph, and it should not contain self-loop and recursive functions.

If a function FA calls the function FB as shown in Figure 1, then the number of hops between these two functions is called the distance from FA to FB. It is written as DISTANCE (FA, FB).

![Figure 1. Function call procedure.](image)

For ∀ u, v ∈ V, DISTANCE (u, v) satisfies
- DISTANCE (u, v) is initialized to 0
- If there are multiple paths from u to v, choose the shortest path
- If u calls v directly then DISTANCE (u, v) = 1
- Generally, DISTANCE (u, v) equals the number of non-key function between u and v plus 1

**Similar Module Extraction Approach**

In this section, we introduce similar module extraction approach. As shows in Figure 2, two malicious samples α and β are preprocessed, and two function call graphs FCG_α and FCG_β are obtained. If the function is not a key function, then it is removed from the FCG. The specific method of removing non-key functions is described in Build Key Function Call Graph. When all non-key functions are removed, two key function call graphs KFCG_α and KFCG_β are generated, respectively.

The function name in KFCG is directly retrieved from smali codes which is a literal name of the function. This literal name is not reliable since it may vary as the malware author’s intention. As a result, it is important to identify the function even though it has a different name. Assuming that a key function in KFCG calls sequences of Android system API F_1, F_2, ..., F_n. These system APIs are joint into a string with comma separated. Then the hash of the joint string is regarded as the hashed name of the key function. When all function names in a KFCG are hashed, hashed key function call graphs HKFCG are generated. In HKFCG, a function could be identified uniquely through its hashed name, then we construct common matrix by extracting common nodes from HKFCG_α and HKFCG_β. The resulting common matrices are written as HKFCM_α and HKFCM_β.
Finally, the similarity is calculated by comparing $HKFCM_\alpha$ and $HKFCM_\beta$ and it is demonstrated in Similarity Calculation.

**Build Key Function Call Graph**

Figure 3 (i) is a function call graph. A circle containing the uppercase letter in the figure indicates a key function and others are non-key functions. The non-key function $a$ is removed at first. Since $A$ calls $a$ and $a$ calls $C$, the hop value $A \rightarrow C$ should be updated to 2 after removing $a$. However, $A$ calls $B$ directly, the hop value of $A \rightarrow B$ is less than the one of $A \rightarrow a \rightarrow B$. Therefore the hop value of $A \rightarrow B$ is not updated. $b$ is removed using a similar method, and the resulting key function call graph $KFCG$ is shown in Figure 3 (ii).

**Similarity Calculation**

Suppose there are two samples $\alpha$ and $\beta$, and their corresponding matrices are $HKFCM_\alpha$ and $HKFCM_\beta$. These two matrices are shown in Eq. 1.

$$HKFCM_\alpha = \begin{bmatrix} C^{\alpha}_{11} & C^{\alpha}_{12} & \cdots & C^{\alpha}_{1i} & \cdots & C^{\alpha}_{1m} \\ C^{\alpha}_{21} & C^{\alpha}_{22} & \cdots & C^{\alpha}_{2i} & \cdots & C^{\alpha}_{2m} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ C^{\alpha}_{i1} & C^{\alpha}_{i2} & \cdots & C^{\alpha}_{ii} & \cdots & C^{\alpha}_{im} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ C^{\alpha}_{m1} & C^{\alpha}_{m2} & \cdots & C^{\alpha}_{mi} & \cdots & C^{\alpha}_{mm} \end{bmatrix}$$

$$HKFCM_\beta = \begin{bmatrix} C^{\beta}_{11} & C^{\beta}_{12} & \cdots & C^{\beta}_{1i} & \cdots & C^{\beta}_{1m} \\ C^{\beta}_{21} & C^{\beta}_{22} & \cdots & C^{\beta}_{2i} & \cdots & C^{\beta}_{2m} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ C^{\beta}_{i1} & C^{\beta}_{i2} & \cdots & C^{\beta}_{ii} & \cdots & C^{\beta}_{im} \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ C^{\beta}_{m1} & C^{\beta}_{m2} & \cdots & C^{\beta}_{mi} & \cdots & C^{\beta}_{mm} \end{bmatrix}$$

The similarity between $\alpha$ and $\beta$ is defined as $S(HKFCM_\alpha, HKFCM_\beta)$ in Eq. 2.
where

\[
f_d \left( C^{ij}_\alpha, C^{ij}_\beta \right) = \begin{cases} 
0 & \text{if } C^{ij}_\alpha = 0 \text{ or } C^{ij}_\beta = 0 \\
1 & \text{if } \left( C^{ij}_\alpha = C^{ij}_\beta \right) \\
\frac{\min(C^{ij}_\alpha, C^{ij}_\beta)}{\max(C^{ij}_\alpha, C^{ij}_\beta)} & \text{otherwise}
\end{cases}
\]

\[
f_s \left( C^{ij}_\alpha, C^{ij}_\beta \right) = \begin{cases} 
0 & \text{if } C^{ij}_\alpha = 0 \text{ and } C^{ij}_\beta = 0 \\
1 & \text{otherwise}
\end{cases}
\]

\[C^{ij}_\alpha\] represents the hop value from \( F_i \) to \( F_j \) in sample \( \alpha \), and \( C^{ij}_\beta \) represents the hop value from \( F_i \) to \( F_j \) in sample \( \beta \). The value of the similarity is between \([0, 1]\), and the higher value indicates the two applications are more similar.

**Experimental Evaluation**

In the experiment, we categorized malware with similar modules in the same class. The threshold of similarity is set to 0.7, and when \( S(HKFCM_\alpha, HKFCM_\beta) \geq 0.7 \), it is considered that the two applications are similar.

The experimental dataset has about 10,500 Android malware. A NANO antivirus is selected to classify the samples, and the classifying result is used as our reference. NANO has categorized the samples into several families; if a family contains more than 450 samples, then it is selected for the experiment. There are eight selected families which are listed in Table 1.

<table>
<thead>
<tr>
<th>Family</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trojan.Android.FakeInst</td>
<td>2858</td>
</tr>
<tr>
<td>Trojan.Android.Agent</td>
<td>2502</td>
</tr>
<tr>
<td>Trojan.Android.Domob</td>
<td>1102</td>
</tr>
<tr>
<td>Trojan.Android.Opfake</td>
<td>1077</td>
</tr>
<tr>
<td>Trojan.Android.Dowgin</td>
<td>1118</td>
</tr>
<tr>
<td>Trojan.Android.WqMobile</td>
<td>925</td>
</tr>
<tr>
<td>Riskware.Android.MobWin</td>
<td>533</td>
</tr>
<tr>
<td>Trojan.Android.Airpush</td>
<td>471</td>
</tr>
</tbody>
</table>

**Similar Module Extraction Rate**

We use the proposed approach to check the extraction of similar modules. The percentage for each family is calculated by the following formula:

\[
\text{Extraction rate} = \frac{\text{Similar module count}}{\text{Total count}} \times 100\%
\]

The extraction rate of each family is listed in Table 2. For instance, our approach returns a total of 1081 malware for Trojan.Android.Domob family. Therefore, the extraction rate for Trojan.Android.Domob is 98.09%.
Table 2. Extraction rate.

<table>
<thead>
<tr>
<th>Family</th>
<th>Extraction rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trojan.Android.FakeInst</td>
<td>82.37%</td>
</tr>
<tr>
<td>Trojan.Android.Agent</td>
<td>69.26%</td>
</tr>
<tr>
<td>Trojan.Android.Domob</td>
<td>98.09%</td>
</tr>
<tr>
<td>Trojan.Android.Opfake</td>
<td>93.22%</td>
</tr>
<tr>
<td>Trojan.Android.Dowgin</td>
<td>97.85%</td>
</tr>
<tr>
<td>Trojan.Android.WqMobile</td>
<td>99.78%</td>
</tr>
<tr>
<td>Riskware.Android.MobWin</td>
<td>99.62%</td>
</tr>
<tr>
<td>Trojan.Android.Airpush</td>
<td>91.93%</td>
</tr>
</tbody>
</table>

Table 3 illustrates the experimental results of similar modules for different families by various antivirus engines, namely F-secure, BitDefender, AhnLab-V3, TrendMicro, Kaspersky, and Avast. Each row in Table 3 represents similar modules which we extracted by our approach. If the antivirus engines classify the malware with a higher percentage in the same family, then it can prove that our method is more efficient for extracting similar modules.

Table 3. Validation of similar modules.

<table>
<thead>
<tr>
<th>Antivirus Engine</th>
<th>Family</th>
<th>F-Secure</th>
<th>BitDefender</th>
<th>AhnLab-V3</th>
<th>TrendMicro</th>
<th>Kaspersky</th>
<th>Avast</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trojan.Android.FakeInst</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99.8%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>Trojan.Android.Agent</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>Trojan.Android.Domob</td>
<td>98.7%</td>
<td>98.9%</td>
<td>99.1%</td>
<td>84.9%</td>
<td>92.3%</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Trojan.Android.Opfake</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>*</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>Trojan.Android.Dowgin</td>
<td>99.8%</td>
<td>99.5%</td>
<td>99.3%</td>
<td>*</td>
<td>99.2%</td>
<td>95.5%</td>
</tr>
<tr>
<td></td>
<td>Trojan.Android.WqMobile</td>
<td>99.6%</td>
<td>99.6%</td>
<td>99.6%</td>
<td>86.0%</td>
<td>98.7%</td>
<td>96.2%</td>
</tr>
<tr>
<td></td>
<td>Riskware.Android.MobWin</td>
<td>98.1%</td>
<td>97.7%</td>
<td>97.9%</td>
<td>93.1%</td>
<td>92.3%</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Trojan.Android.Airpush</td>
<td>96%</td>
<td>96.2%</td>
<td>89.2%</td>
<td>*</td>
<td>94.4%</td>
<td>*</td>
</tr>
</tbody>
</table>

*indicates the number of detected malware are less than 20 which are not taken into consideration.

**Time Complexity**

We assume that two malware have \( m \) and \( n \) functions. To evaluate the time complexity, \( mn \) comparisons are conducted based on code similarity [12]. In Android applications, the proportion of non-key functions is higher than that of the key functions. Statistics on the experimental samples say that the proportion of non-key functions is about 70.75%. Therefore, the time complexity for the proposed approach is \( (1 - 0.7075)^2 \times mn = 0.086 \times mn \).

**Conclusion**

This paper presents an approach that can extract similar modules from Android malware. The proposed approach applies Android system APIs sequences to build key function call graph for extracting similar modules. The experimental results demonstrate that the proposed approach has high accuracy and acceptable performance. Moreover, the results indicate that many malware are not written from scratch, and they often borrow codes from other malware to reduce the burden of creating new malware.
Acknowledgment

This research work was supported by Guangdong National Natural Science Foundation of China under Grant No. U1401258, and Shenzhen Technology Development under Grants No.CXZZ20150813155917544 and JSGG20151117110411567.

References


