Kullback-Leibler Divergence Based Detection of Repackaged Android Malware

Hossain Shahriar, Victor Clincy  
Kennesaw State University  
United States  
hshahria@kennesaw.edu, vclincy@kennesaw.edu

ABSTRACT: Android applications are widely used by millions of users to perform many activities. Unfortunately, legitimate and popular applications are targeted by malware authors and they repackaged the existing applications by injecting additional code intended to perform malicious activities without the knowledge of end users. Thus, it is important to validate applications for possible repackaging before their installation to safeguard end users. This paper presents the detection of repackaged malware application based on Kullback-Leibler Divergence (KLD) metric. Our approach builds the population distribution of a legitimate and suspected repackaged malware application based on a set of Smali opcode. A high KLD value indicates that an application is dissimilar compared to an original application, hence likely a repackaged application. The approach has been validated based on real-world malware samples and repackaging them to a legitimate application. The results indicate that KLD values remain high for all the malware when repackaged within a legitimate application, and hence can be used as a suitable metric for detection of new malware.

Keywords: Android Malware, Repackaging, Decomplier, Kullback-leibler Divergence, Smali Opcode, Information Theory

Received: 10 November 2014, Revised 13 December 2014, Accepted 18 December 2014

© 2015 DLINE. All Rights Reserved

1. Introduction

Android is an open source operating system for mobile devices. Currently Android occupies close to 80% of mobile device market share [1]. We now highly dependent on Android devices as well as the applications that run on the platform. In particular, many useful activities such as phone call, message sending, and game playing are performed with applications. End users rely on Android market to obtain legitimate applications and install in their devices.

Unfortunately, popular Android applications are becoming the target of malware authors. In particular, there exists available open source tools that can be used to download legitimate applications, disassemble these applications, insert with additional malicious code intended to perform unauthorized activities, repackage the modified applications, and finally lure or distribute to potential victims to download the applications and install in their devices. The modified and repackaged application if installed by a victim in his/her phone, unwanted malicious activities take place without his/her knowledge.

A detailed study of a large set of malware applications and characteristics revealed that most of the well-known malware samples belong to few popular legitimate applications available in the market [2]. In particular, three common types of malicious
A large number of literature works have recently addressed Android malware from the perspective of classification [6, 7, 8] and detection [9-19] based on anomalous activities or permission sets present in suspected applications. However, very few research works [22, 23, 24] have addressed the issue of detecting repackaged Android malware applications considering the availability of legitimate applications. Existing approaches statically analyzes the assembly code. These approaches suffer from false positive warnings or generating numerous hashes based on opcode sets resulting in significant computation time. In this paper, we propose metric-based detection of repackaged malware. In particular, we apply a popular measurement brought from information theory called Kullback-Leibler Divergence (KLD) [25].

A repackaged application when compared to its original version of the application is different compared to the set of available functionalities. We capture the difference by proposing a set of opcode (Smali) to compute needed population sets as part of KLD. Our proposed opcodes consider the existing knowledge from malware detection domain such as method call invocation that may send a message sneakily as part of malware application functionality. To perform computation for missing elements of a population set, we rely on back-off smoothing algorithms.

The approach has been performed for a number of malicious and repackaged android applications and compared with the known legitimate version of the applications. The results seem promising, KDL values between malware and known good application differs significantly.

The paper is organized as follows. Section II provides an overview of Android application packaging and related work. In Section III, the proposed approach is discussed in details. Section IV discusses the experimental evaluation. Section V draws the conclusions and discusses future work.
2.2 Related Works on Android Malware Detection

Crusselle et al. [22] detect repackaged applications by computing the data dependency graph (DDG) statically for each of the methods statically. The graphs are compared for similarity for a known application to identify possible deviation due to additional methods part of a repackaging malware application.

Li et al. [23] propose feature hashing-based technique. They first identify k-grams of various opcode sequence patterns within each basic block and consider them as features. The presence or absence of each features in dex files are encoded in a vector. All vectors obtained from files are merged to obtain the fingerprint of each application.

Zhou et al. [24] computes hash values for each local unit of opcode sequence of the classes (dex files). The long opcode is handled by splitting into small units and computing hashes for each split unit. Finally, all individual hashes are combined into one hash values. This way any additional inserted code is detected not only for the overall application, but locating the files or specific instruction sets that are inserted by malware authors. The approach suffers from false positive if inserted dummy opcode does not have any negative impact.

In contrast to these efforts, our proposed technique relies on metrics and computes them at application runtime in sandbox.

We are aware of several works that classify malware applications and their detection technique. Amamra et al. [7] perform a survey on malware detection approaches highlighting two broader classes of malware detection: signature and anomaly-based. Porter et al. [6] performed a survey on malicious characteristics for mobile device malware in 2011. Cooper et al. [8] classify android malware detection techniques and comparatively identify the advantages and disadvantages including static analysis, sandboxing, and machine learning approaches. Tanh et al. [29] characterize malware and demonstrate what end users can do to check the presence of malware and prevent them.

Enck et al. [9] analyzed a large set of android applications and identified dataflow, structure, and semantic patterns. The dataflow patterns identify whether any sensitive data information piece should not be sent to outside (e.g., IMEI, IMSI, ICC-ID). Enck et al. [10] proposed a rule-based certification technique to check the presence of undesirable properties in applications suspected as malware. The approach starts from general functionality requirements and then analyze whether required permissions can create conflicting operations that are used in malware operations. Batyuk et al. [11] perform static analysis on binary code of android applications (after decompressing APK and decoding Java bytecode into Smali assembly language. They look for the presence of APIs that may be relevant of reading sensitive information (e.g., IMEI or device identifier, IMSI or subscriber identifier, phone number, writing information to output stream). Yang et al. [16] detect money stealing malware by examining the manifest file of android applications to see if billing permission is present. They look for specific method calls or APIs that perform SMS messaging or calls to premium phone numbers. Permission files are analyzed in some approaches as part of malware mitigation. Barrera et al. [12] apply selforganizing map-based learning to cluster permission sets. The study and findings cannot be suitably applied for detecting malware as both malicious and benign applications may have similar type of permissions. Similarly, Felt et al. [13] compared the permission system between Google Chrome and Google Android, and performed a subjective analysis for improving permission model in general for security and user level awareness.

Nevertheless, detection technique of repackaged malware is still needed to identify malicious behaviors of malware, and our approach is complementary to these earlier efforts.

Several works rely on live analysis of applications running in a sandbox environment. Enck et al. [14] analyze the dataflow of Android application to detect privacy leak (whether sensitive data are being transferred to third parties related to advertisement services). Blasing et al. [15] also develop a sandbox to perform dynamic analysis of suspected applications in an isolated environment. They first perform static analysis to identify suspected APIs such as loading of a library method and class, retrieving a list of directory, issuing a system level command like file deletion. The sandbox environment is used to launch necessary activity to confirm those behaviors as part of malware detection. All these earlier efforts are complementary to our proposed approach intended to build defense in-depth against repackaged malware.

3. Proposed Detection Approach

3.1 Kullback-Leibler Divergence (KLD) Computation

The Kullback-Leibler Distance (KLD) computes the divergence or distance between two given probability distributions. Let us
assume that $P$ and $Q$ represent two probability distributions, where $P = \{p_1, \ldots, p_n\}$ and $Q = \{q_1, \ldots, q_n\}$. Then, the KLD is defined as follows [25]:

$$KLD (P, Q) = \sum_i p_i \cdot \log_2 \left( \frac{p_i}{q_i} \right) \ldots (i)$$

Here, the following two constraints (Equations (ii) and (iii)) are satisfied:

$$\sum_i p_i = 1 \ldots (ii)$$
$$\sum_i q_i = 1 \ldots (iii)$$

The KLD can be viewed as the additional message-length required when using a code based on the target distribution ($Q$) compared to using a code based on the true distribution ($P$). Therefore, KLD is also denoted as the relative entropy between $P$ and $Q$ in information theory. Note that KLD is not symmetric (i.e., $KLD (P, Q) \neq KLD (Q, P)$). Also, $KLD (P, Q) = 0$, iff $P = Q$.

We start with a hypothesis that the Kullback-Leibler Divergence (KLD) between an original and repackaged malware application should be a high number. On the other hand, the KLD among 2 legitimate applications from the same source, KLD value should be very low.

To compute the KLD between two population sets (or probability distributions) need to be defined at the beginning. We focus on a set of well-known opcode that may be common in both legitimate and repackaged malware applications. A set of opcode elements are extracted from a known legitimate application to build $P$ set. Now, given that we have a new application ($Q$), we extract the similar opcode occurrence probability distribution and compute the divergence to detect possible repackaged application.

The challenge of computing $KLD (P, Q)$ is the term $p_i \cdot \log_2 \left( \frac{p_i}{q_i} \right)$. It can be rewritten as subtraction of two terms: $p_i \cdot \log_2 (p_i) - p_i \cdot \log_2 (q_i)$. If $p_i$ or $q_i$ is zero (no occurrence of a specific opcode is observed), then the term becomes infinite, which results in $KLD (P, Q)$ to be infinite as well.

<table>
<thead>
<tr>
<th>Name</th>
<th>Smalio opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>invoke-super {parameter},methodtocall</td>
<td>Invokes the virtual method of the immediate parent class</td>
</tr>
<tr>
<td>$f_2$</td>
<td>invoke-static {parameters},methodtocall</td>
<td>Invokes a static method with parameters.</td>
</tr>
<tr>
<td>$f_3$</td>
<td>invoke-direct {parameter},methodtocall</td>
<td>Invokes a method with parameters without the virtual method resolution</td>
</tr>
<tr>
<td>$f_4$</td>
<td>invoke-virtual {parameter},methodtocall</td>
<td>Invokes a virtual method with parameters.</td>
</tr>
<tr>
<td>$f_5$</td>
<td>const-string vx,string_id</td>
<td>Puts reference to a string constant identified by string_id into vx.</td>
</tr>
<tr>
<td>$f_6$</td>
<td>new-instance vx,type</td>
<td>Instantiates an object type and puts the reference of the newly created instance into vx</td>
</tr>
</tbody>
</table>

Table 1. Description of opcode for population set
To address this issue, we propose to apply a well-known smoothing technique known as constant back-off [26]. Here, all zero probability values in both \( P \) and \( Q \) are replaced with a very negligible constant probability value and all the nonzero values are equally subtracted with the same constant value proportionally so that Equations (ii) and (iii) are still satisfied. This simple step results in two smoothed probability distributions that we denote as \( P' \) (derived from \( P \)) and \( Q' \) (derived from \( Q \)). So, we essentially compute \( KLD(P', Q') \) to avoid infinity problem instead of \( KLD(P, Q) \).

### 3.2 Elements of Population Set for KLD Computation

Table 1 shows a set of Smali opcode that we propose to build population elements (\( f_1 - f_6 \)). We consider six opcode: `invokesuper`, `invoke-static`, `invoke-direct`, `invoke-virtual`, `conststring`, and `new-instance`. A description of the opcode is provided in Table 1.

For example, `invoke-direct` opcode invokes a method specified in the second argument while supplying a set of parameter specified in the first argument. We choose these opcodes based on the literature knowledge that injected code by malware authors are mostly doing a set of operations such as sending of SMS messages to premium numbers, probing device id and

```
public class HelloWorldActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView text = new TextView(this);
        text.setText("Hello World, Android");
        setContentView(text);
    }
}
```

Figure 1. Example Java code for legitimate application (P)

```
...# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
    new-instance v0, Landroid/widget/TextView;
    invoke-direct {v0, p0}, Landroid/widget/TextView;->(Landroid/content/Context;)V
    .local v0, text:Landroid/widget/TextView;
    const-string v1, "Hello World, Android"
    invoke-virtual {v0, v1}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
    invoke-virtual {p0, v0}, Lcom/test/helloworld/HelloWorldActivity;->setContentView(Landroid/view/View;)V
...end method
```

Figure 2. Example opcode for the legitimate application (P)
sending the information over the network. All these operations require method call invocation as well as often defining constant string values (const-string opcode) that may store attacker supplied information such as phone number.

3.3 Example of repackaged Malware Detection
We consider a legitimate example of Android application that is intended to display a simple message “hello world” in an Activity class as shown in Figure 1. In this example, the onCreate() method displays the message by accessing the TextView object and invoking the setText() method call.

```java
public class HelloWorldActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView text = new TextView(this);
        text.setText("Hello World, Android");
        setContentView(text);
        SmsManager smsManager = SmsManager.getDefault();
        String phone = "1-900-222-3333";
        smsManager.sendTextMessage(phone, null, "sms", null, null);
    }
}
```

Figure 2 shows a snapshot the Smali opcode that can be obtained based on a suitable reverse engineering tool such as apktool [21]. We display and highlight the relevant opcode as part of population set (e.g., invoke-super) for the onCreate() method call only due to space limitation.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Occurrence (p)</th>
<th>Smoothed (p_j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke-super (f_1)</td>
<td>1/4</td>
<td>1/4 - e/5</td>
</tr>
<tr>
<td>invoke-static (f_2)</td>
<td>0/4</td>
<td>2e/4</td>
</tr>
<tr>
<td>invoke-direct (f_3)</td>
<td>1/4</td>
<td>1/4 - e/5</td>
</tr>
<tr>
<td>invoke-virtual (f_4)</td>
<td>2/4</td>
<td>2/4 - 2e/5</td>
</tr>
<tr>
<td>const-string (f_5)</td>
<td>1/4</td>
<td>1/4 - e/5</td>
</tr>
<tr>
<td>new-instance (f_6)</td>
<td>0/4</td>
<td>2e/4</td>
</tr>
</tbody>
</table>

Table 2. Occurrence of population element from legitimate application (P)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Occurrence (q)</th>
<th>Smoothed (q_j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke-super (f_1)</td>
<td>1/8</td>
<td>1/8 - e/8</td>
</tr>
<tr>
<td>invoke-static (f_2)</td>
<td>1/8</td>
<td>1/8 - e/8</td>
</tr>
<tr>
<td>invoke-direct (f_3)</td>
<td>1/8</td>
<td>1/8 - e/8</td>
</tr>
<tr>
<td>invoke-virtual (f_4)</td>
<td>2/8</td>
<td>2/8 - 2e/8</td>
</tr>
<tr>
<td>const-string (f_5)</td>
<td>3/8</td>
<td>3/8 - 3e/8</td>
</tr>
<tr>
<td>new-instance (f_6)</td>
<td>0/8</td>
<td>e</td>
</tr>
</tbody>
</table>

Table 3. Occurrence of population element from repackaged malware application (Q)
Let us assume that a malware author injects an SMS message sending operation to a premium number right after the hello world message display operation. The added Java code is shown and highlighted in Figure 3. Here, the SmsManager object is first retrieved followed by invocation of the sendTextMessage() method call having five arguments including a premium phone number (1-900-222-3333) and a message (sms). Figure 4 shows the Smali opcode for the malware activity of sending a SMS message to a premium number with the population elements highlighted (e.g., const-string, invoke-virtual).
Based on Figure 2, we develop the occurrence probability of the population element to build P set as follows in Table 2. We also show the smoothed probability values due to missing elements \( f_2, f_4 \). Here, we assume \( e \) is a very small number having the value of 0.00001. Similarly Table 3 computes Q set based on the repackaged application opcode with necessary smoothing.

Now, Table 4 shows the detailed steps of computing KLD \( (P', Q') \). The last row shows the value of KLD as 0.85367, which we can consider high.

### 4. Evaluation

We evaluated our approach by using a set of malware samples obtained from the authors of [28] (Malgenomre project dataset). The same benchmark has been widely used for related research work as well. From the benchmark, we randomly choose samples from three malware families: DroidDream, Gone60, and Plankton. Table 5 shows brief characteristics of the malware family along with number of samples we evaluate from each of the families. For example, DroidDream malware attempts to obtain product ID, device type, language, country, and send them to a remote server via text message.

### Table 4. Computation of KLD \((P', Q')\)

<table>
<thead>
<tr>
<th>Element (i)</th>
<th>( p'_i )</th>
<th>( q'_i )</th>
<th>( \log_2(p'_i) )</th>
<th>( \log_2(q'_i) )</th>
<th>( p'_i \log_2(p'_i/q'_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke-super ( f_i )</td>
<td>0.25000</td>
<td>0.12500</td>
<td>-2.00001</td>
<td>-3.00001</td>
<td>0.25000</td>
</tr>
<tr>
<td>invoke-static ( f_i )</td>
<td>0.00001</td>
<td>0.12500</td>
<td>-17.60964</td>
<td>-3.00001</td>
<td>-0.00007</td>
</tr>
<tr>
<td>invoke-direct ( f_i )</td>
<td>0.25000</td>
<td>0.12500</td>
<td>-2.00001</td>
<td>-3.00001</td>
<td>0.25000</td>
</tr>
<tr>
<td>invoke-virtual ( f_i )</td>
<td>0.50000</td>
<td>0.25000</td>
<td>-1.00001</td>
<td>-2.00001</td>
<td>0.50000</td>
</tr>
<tr>
<td>const-string ( f_i )</td>
<td>0.25000</td>
<td>0.37500</td>
<td>-2.00001</td>
<td>-1.41505</td>
<td>-0.14624</td>
</tr>
<tr>
<td>new-instance ( f_i )</td>
<td>0.00001</td>
<td>0.000001</td>
<td>-17.60964</td>
<td>-16.60964</td>
<td>-0.00001</td>
</tr>
<tr>
<td>Sum ( ( p'_i \log_2(p'_i/q'_i) ))</td>
<td>0.85367</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 5. Occurrence of population element from repackaged malware application \((Q)\)

<table>
<thead>
<tr>
<th>Malware family</th>
<th>Characteristics</th>
<th># of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DroidDream</td>
<td>• Steal IMEI, IMSI, and phone number&lt;br&gt;• Send information to remote server via SMS messages</td>
<td>16</td>
</tr>
<tr>
<td>Gone60</td>
<td>• Steal IMEI, IMSI and phone number&lt;br&gt;• Steal phone’s state: calls log, SMS, contacts, account&lt;br&gt;• Send information to remote server via SMS messages</td>
<td>9</td>
</tr>
<tr>
<td>Plankton</td>
<td>• Change or copy file in external storage&lt;br&gt;• Download and install apps&lt;br&gt;• Stolen location information: GPS, Google, Country code&lt;br&gt;• Send information to remote server via SMS messages</td>
<td>11</td>
</tr>
</tbody>
</table>
We first use the apktool to disassemble the obtained samples. Figure 5 shows an example snapshot of deassembling using the apktool. We then search if original application for a given malware family is available in the Google market place or any sources or not. Since all the legitimate applications infected with the three malware malware have been removed already from market place, we build a small benign Android application (tip calculator that computes tip amount based on user supplied inputs). We disassemble it and compute the $P$ set. We then use the same apktool and inject the malware classes and add needed opcode to trigger the classes, and then compute $Q$ set from the new application package. We implement a Java class to automate the computing of population set occurrence probability along with KLD computation.

![Figure 5. Example run of apktool for decompiling malware](image)

Figure 6 shows an example of Smali code from a sample of Plankton malware application. Here, the opcode `invokevirtual` (highlighted) are used to read information from an input stream and send it to an output file stream, which was part of downloading application from network stream to a local storage of the Android device.

```
    .local v0, "b":[B
    invoke-virtual {v3, v0}, Ljava/io/InputStream;->read([B)I
    .line 164
    invoke-virtual {v2, v0}, Ljava/io/FileOutputStream;->write([B)V
```

![Figure 6. An example of Smali code from a Plankton sample](image)

We now discuss the obtained results. Figure 7 shows the histogram representing the population element set occurrence for $P$.

![Figure 7. Population element distribution for P](image)
Figure 8 shows the population element distribution ($f_i - f_o$) for all 16 samples of the DreamDroid family after they are injected into $P$. For each of the repackaged applications, the occurrence of population elements ($f_i - f_o$) increases significantly compared to the legitimate application.

Figure 8. Population element distribution of DreamDroid samples (Q)

Figure 9. KLD values for various DreamDroid samples

Figure 10. KLD values for Plankton family samples
Figure 9 shows KLD \((P, Q)\) values as part of detecting repackaged applications having *DreamDroid* malware. We rename 16 different malware samples from \(da-dp\) for reader’s convenience. Note that the highest KLD we observed was for the de sample (5.3382) and the lowest value was for the dd sample (0.1003). Similarly, Figures 10 and 11 display the obtained KLD values as we detect the repackaged malware from *Plankton* and *Gone60* family. The highest and lowest KLD values for Plankton family were 0.01968 and 6.1368, respectively. The highest and lowest KLD values for *Gone60* family were 0.01484 and 4.1396, respectively.

From the obtained data, a KDL threshold value can be chosen to detect new repackaged application. We suggest choosing KLD values above zero.

### 5. Conclusions and Future Work

Repackaging tools are now highly available in today’s application development market and malware authors are taking full advantages by downloading legitimate Android applications, injecting additional code, repackaging and luring potential victims to install the modified applications in their devices. This paper proposes a metric-based approach using Kullback-Leibler Distance (KLD) to identify altered legitimate applications and warn users for possible repackaging. We propose a set of population element features to compute the occurrence probability of specific Smali opcode that may indicate likely malicious activities such as method call invocation performing a message sending operation. The approach has been evaluated with a set of sample malware applications from a real-world benchmark suite and the initial obtained results look promising. For all the repackaging cases, we find that the KLD value exceeds zero and a higher number frequently appears. Our approach can provide the footstep for developing more metric-based solution to combat against repackaged malware detection.

Our future works remains to include more elements for population building set as well as validating with more sample malwares. We also plan to develop a network-based online detection approach to validate whether an application is repackaged or not for a given legitimate application based on other suitable metrics from information theoretic and data compression domains.

### References


