Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis

Yu Feng, Saswat Anand, Isil Dillig and Alex Aiken

FSE 2014

Presented by ke tian
Outlines:

• What is the problem?
  identify Android malware

• What is the solution/contribution?
  signature based specification with graph assistant (ICCG)

• How efficient is the solution?
  low false positive+ false negative
Android attacks

Repackaging
86.0% of # apps in Genome

Trojan
Stealing/sniffing

(a) The Update Dialogue
Figure 2. An Update Attack from BaseBridge
(b) Installation of A New Version

[1]Dissecting Android Malware: Characterization and Evolution
Android Components

1. **Activities**
   - Single screen of a UI
   - Each activity is completely isolated from each other
   - Different app can invoke activity of another app, if permitted
   - Runs in background

2. **Services**
   - Perform long running operation
   - Perform remote processes
   - No user interface

3. **Content providers**
   - Shared set of application data
   - Filesystem, SQLite, web etc
   - Used for alerts

4. **Broadcast receivers**
   - Can use status bar for notification
   - No user interface

http://mithileshjoshi.blogspot.com/2015/06/what-is-android-application-components.html
Communications in Android

1. startActivity()
2. Android System
3. oncreate()
ICCG (inter-component call graph)

Node: component name

Edge: 1)component A --> component B
2)system events

System event
intent

Activity
Receiver
Service
In App
1. GDEvent(SMS_RECEIVED).
2. GDEvent(NEW_OUTGOING_CALL).
3. GoldDream :- receiver(r),
   icc(SYSTEM, r, e, _), GDEvent(e),
   service(s), icc*(r, s),
   flow(s, DeviceId, s, Internet),
   flow(s, SubscriberId, s, Internet).

Figure 2: GoldDream signature (simplified)
ICCG → Signature

Spec Language → Static Analysis

Taint Analysis
Malware Spec Language

Purpose: use a languages/semantics to describe the app’s inner-property/behavior

- Component type predicates: service(c)
- Predicate icc: icc*(p,q)  <example>
- Predicate calls:  calls(c,m)
- Predicate flows:  flow(p, so, q, si)
Initialize an ICC through an intent

```java
public class MainAct extends Activity {
    protected void onCreate(Bundle b) {
        foo();
        bar();
    }

    void foo() {
        Intent i = new Intent();
        i.setAction(android.intent.action SEND);
        i.setType("text/plain");
        startActivity(i);
    }

    void bar() {
        Intent n = new Intent();
        n.setClass(MsgAct.class);
        startActivity(n);
    }
}
```

Figure 3: ICC example.

```
<activity android:name="MsgAct">
    ...
</activity>
```

Figure 4: A snippet of AndroidManifest.xml

Q: Explicit intent or implicit intent?

Icc(MainAct,MsgAct,SEND,text/plain)
Static Analysis

- Call graph construction
  - conventional approach
  - pointer analysis (heap object)
- Data flow Analysis for intents <example>
  - intent analysis (intent filters)
  - transfer functions (complex algebra)
- Construct the ICCG
  - define construction rules (algebra)
1. public class MainAct extends Activity {
2.     protected void onCreate(Bundle b) {
3.         foo();
4.         bar();
5.     }
6.     void foo() {
7.         Intent i = new Intent();
8.         i.setAction(android.intent.action.SEND);
9.         i.setType("text/plain");
10.        startActivity(i);
11.    }
12.    void bar() {
13.        Intent n = new Intent();
14.        n.setClass(MsgAct.class);
15.        startActivity(n);
16.    }
17. }
18. public class MsgAct extends Activity { ... }

Figure 3: ICC example.
Taint Analysis

1. //Source annotation in android.telephony.TelephonyManager
2. @Flow(from="$getDeviceId",to="/return")
3. String getDeviceId(){ ... }

7. //Sink annotation in android.telephony.SmsManager
8. @Flow(from="text",to="/sendTextMessage")
9. void sendTextMessage(...,String text,...){ ... }

10. //Transfer annotation in java.lang.String
11. @Flow(from="this",to="/return")
12. @Flow(from="s",to="/return")
13. String concat(String s){ ... }

If para S is tainted, Then, @return is tainted

Figure 9: Source, Sink and Transfer annotations.
Taint Analysis

\[
\frac{\text{src}(m_i, l), \ m_i \rightarrow o}{\text{tainted}(o, l)} \quad \text{(Source)}
\]

\[
\frac{\text{tainted}(o_1, l), \ m_i \rightarrow o_1, \ m_j \rightarrow o_2}{\text{transfer}(m_i, m_j)} \quad \text{(Transfer)}
\]

\[
\frac{\text{tainted}(o_2, l)}{\text{tainted}(o, so), \ m_i \rightarrow o, \ \text{sink}(m_i, si)} \quad \text{(Sink)}
\]

\[
\frac{\text{flow}(so, si)}{}
\]

Static taint analysis (complex algebra)
Taint Analysis

public class ListDevice extends Activity {
    protected void onCreate(Bundle bd) {
        1. Device n,m;
        2. ... 
        3. String x = "deviceId=";(O1)
        4. String y = TelephonyManager.getDeviceID();(O2)
        5. String z = x.concat(y);(O3)
        6. m.f = z;
        7. n = m;
        8. String v = n.f;
        9. smsManager.sendTextMessage("3452",null,v,null,null);
    }
}

Figure 5: Example illustrating data flow
## Results

### Table 6: Examples of Apposcopy’s signatures.

<table>
<thead>
<tr>
<th>Malware family</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADRD</td>
<td>ADRD :- receiver(r), icc(SYSTEM, r, BOOT_COMPLETED, _), receiver(s), service(t), icc(r, s), icc(s, t), icc(t, s), flow(t, DeviceId, t, ENC), flow(t, SubscriberId, t, ENC), flow(t, ENC, t, Internet).</td>
</tr>
<tr>
<td>BeanBot</td>
<td>BeanBot :- receiver(r), service(s), service(t), service(q), icc(SYSTEM, r, PHONE_STATE, _), calls(r, abortBroadcast), icc(r, s), icc(s, t), icc(s, q), flow(s, DeviceId, s, Internet), flow(s, Line1Number, s, Internet), flow(s, SimSerialNumber, s, Internet).</td>
</tr>
<tr>
<td>CoinPirate</td>
<td>CoinPirate :- receiver(r), receiver(t), icc(SYSTEM, r, SMS_SENT, _), icc(SYSTEM, r, SMS_RECEIVED, _), service(s), calls(r, abortBroadcast), calls(s, sendTextMessage), icc(r, s), icc(s, t), flow(s, DeviceId, s, Internet), flow(s, SubscriberId, s, Internet), flow(s, Model, s, Internet), flow(s, SDK, s, Internet).</td>
</tr>
</tbody>
</table>
### Results

FN = A belongs to family F but Apposcopy cannot detect

FP = A does not belong to family F but Apposcopy wrongly identifies

**Why FN >> FP?**

<table>
<thead>
<tr>
<th>Malware Family</th>
<th>#Samples</th>
<th>FN</th>
<th>FP</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>DroidKungFu</td>
<td>444</td>
<td>15</td>
<td>0</td>
<td>96.6%</td>
</tr>
<tr>
<td>AnserverBot</td>
<td>184</td>
<td>2</td>
<td>0</td>
<td>98.9%</td>
</tr>
<tr>
<td>BaseBridge</td>
<td>121</td>
<td>75</td>
<td>0</td>
<td>38.0%</td>
</tr>
<tr>
<td>Geinimi</td>
<td>68</td>
<td>2</td>
<td>2</td>
<td>97.1%</td>
</tr>
<tr>
<td>DroidDreamLight</td>
<td>46</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>GoldDream</td>
<td>46</td>
<td>1</td>
<td>0</td>
<td>97.8%</td>
</tr>
<tr>
<td>Pjapps</td>
<td>43</td>
<td>7</td>
<td>0</td>
<td>83.7%</td>
</tr>
<tr>
<td>ADRD</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>jSMSHider</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>DroidDream</td>
<td>14</td>
<td>1</td>
<td>0</td>
<td>92.9%</td>
</tr>
<tr>
<td>Bgserv</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>BeanBot</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>GingerMaster</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>CoinPirate</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td>DroidCoupon</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100.0%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1027</td>
<td>103</td>
<td>2</td>
<td>90.0%</td>
</tr>
</tbody>
</table>
Q&A

Why semantic-based?

Can apposcopy identify malicious/benign behavior? No

Signature resistance? Code reordering/code injection/code rewriting