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

by Feng et al [FSE ‘14]

presented by Maaz Ahmad
The Malware Problem


Nearly half of Android Malware attempt to steal personal data.

Kaspersky Lab detected 29,695 new malware modifications in a quarter of a year.


Prevalent solutions

Taint Analysis;
- Information flow analysis
- Expose applications that leak confidential data
- Not all applications that leak data are malware
- Security audit required to filter benign applications from malware

Signature Based Detectors;
- Pattern matching technique, searches for specific instruction or byte sequences
- Great against known malware
- Only as good as their signature database (which must be kept up to date)
- Easy to work around by introducing code transformations
What we need

Tools that operate automatically
- No security audit required

Tools that are smart
- Can look past minor program obfuscations
- Can adapt to new unknown malware
Apposcopy: a best of both worlds?

Semantic based approach for malware that steal information

Two main components:
- A high level language to describe semantic signatures of malware
  - Control flow properties (eg: broadcast receiver launches a service)
  - Data flow properties (eg: reads contacts data and sends it through SMS)
- A powerful static analysis for deciding if an application matches the a signature
  - Inter-component callgraph (ICCG) for control flow analysis
  - Taint analysis for data flow

High level signatures are resistant to low level code transformations
An Example: GoldDream Malware

A family of malware software that

- Spies on user’s messages and calls
- Registers a receiver to listen for these events
- Once invoked, starts a background service w/o users knowledge
- Uploads call and SMS data to remote server
- Uploads other personal data such as IMEI number, subscriber ID etc.
GoldDream Signature

GDEvent(SMS RECEIVED).
GDEvent(NEW OUTGOING CALL).
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)
Figure 1: Partial ICCG for an instance of the GoldDream malware family
Signature Detection (Taint Analysis)

from sjgo.client.zjService:
    $getSimSerialNumber -> !INTERNET
    $getDeviceId -> !INTERNET
    $getSubscriberId -> !INTERNET
    $getDeviceId -> !sendTextMessage
    $getSubscriberId -> !sendTextMessage

from boy.android.game.fiveInk.FiveLink:
    $ID -> !INTERNET
    $MODEL -> !INTERNET

get.youmi.android.AdActivity:
    $getDeviceId -> !WebView
    $ExternalStorage -> !WebView
Malware Spec Language

Datalog program augmented with built in predicates
A predicate must be defined for each malware family
Helper predicates may be defined
Datalog

Each program comprises of:

- A set of facts
  - \texttt{parent("Bill", "Mary")}
  - \texttt{GDEvent(SMS RECEIVED)}
- A set of rules
  - \texttt{ancestor(x, y) :- parent(x, z), ancestor(z, y)}

Predicates may contain variables, constants or “\_” (meaning: don’t care)
Predicates represent relations
Built-in Predicates

Component type predicates

Inter-component communication predicates

Predicate $calls()$

Predicate $flows()$
Component type predicates

Represent different kinds of components in the Android framework:
- service(c)
- activity(c)
- receiver(c)
- contentprovider(c)

Used to establish type of c
Correspond to relation of type (component : C)
ICC Predicates

Inter-component communication predicates

ICC in Android revolves around Intents

Methods that take Intent as parameter are called ICC methods

Instructions that invoke ICC Methods are called ICC sites

When ICC is initiated, life-cycle methods of the target component are invoked

<table>
<thead>
<tr>
<th>Activity</th>
<th>startActivity(Intent)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>startActivityForResult(Intent, int)</td>
</tr>
<tr>
<td></td>
<td>startActivityIfNeeded(Intent, int)</td>
</tr>
<tr>
<td></td>
<td>startNextMatchingActivity(Intent)</td>
</tr>
<tr>
<td>Service</td>
<td>startService(Intent)</td>
</tr>
<tr>
<td></td>
<td>bindService(Intent)</td>
</tr>
<tr>
<td>BroadcastReceiver</td>
<td>sendBroadcast(Intent)</td>
</tr>
<tr>
<td></td>
<td>sendBroadcast(Intent, String)</td>
</tr>
<tr>
<td></td>
<td>sendOrderedBroadcast(Intent, String)</td>
</tr>
</tbody>
</table>

Table 2: A partial list of life-cycle APIs

<table>
<thead>
<tr>
<th>Activity</th>
<th>onCreate(Bundle), onRestart(), onStart(), onResume(), onPause(), onStop(), onDestroy()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>onCreate(), onBind(Intent)</td>
</tr>
<tr>
<td></td>
<td>onStartCommand(Intent, int, Intent)</td>
</tr>
<tr>
<td>BroadcastReceiver</td>
<td>onReceive(Context, Intent)</td>
</tr>
</tbody>
</table>
Intents passed to target may carry many types of information
Apposcopy only considers ‘action’ and ‘data’
CC predicate represents inter-component communication in Android framework
\[ \text{icc}(s,t,a,d) \]
• Corresponds to relation of type \((\text{source} : S, \text{target} : T, \text{action} : A, \text{data} : D)\)
• A and D may be \(\bot\)
Definition 3.1: Target of any ICC site is all components that receive passed intent in some execution of the program.

Definition 3.2: $m_1 \rightarrow m_2$, if method $m_1$ directly calls $m_2$. $m_1 \rightarrow^* m_2$ if $m_1$ transitively calls $m_2$.

Definition 3.3: The predicate $icc(s,t,a,d)$ is true iff:
- $m_1$ is a lifecycle method of $s$
- $m_1 \rightarrow^* m_2$
- $m_2$ contains an $icc$ site with target $t$
- The action and data values are $a$ and $d$ respectively

Definition 3.4: $icc^*(s,t)$ is true if $s$ transitively communicates with $t$.
- $icc^()$ allows the signatures to be more robust to code alterations
Predicate calls()

Represents a method call by a component
Corresponds to the type \((\text{component} : \text{C}, \text{callee} : \text{M})\)
calls\((c, m)\) is true iff:
- \(n\) is a life-cycle method defined in component \(c\)
- \(n \rightarrow^* m\)

Help detect malware that abuse Android API methods

Table 3: A non-exhaustive list of Android methods that are candidates of abuse

<table>
<thead>
<tr>
<th>Operation &amp; Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BroadcastReceiver: void abortBroadcast()&gt;</td>
</tr>
<tr>
<td>Block current broadcaster.</td>
</tr>
<tr>
<td>&lt;Runtime: Process exec(java.lang.String)&gt;</td>
</tr>
<tr>
<td>Execute a command.</td>
</tr>
<tr>
<td>&lt;System: void loadLibrary(java.lang.String)&gt;</td>
</tr>
<tr>
<td>Perform native call.</td>
</tr>
<tr>
<td>&lt;PackageManager: List getInstalledPackages()&gt;</td>
</tr>
<tr>
<td>Get all application packages.</td>
</tr>
<tr>
<td>&lt;DexClassLoader: void &lt;init&gt;(...,ClassLoader)&gt;</td>
</tr>
<tr>
<td>Load classes from .jar and .apk files.</td>
</tr>
</tbody>
</table>
Predicate flows()

Represents data flow to help detect sensitive information leak

**Definition 3.5:** Source and sink variables are annotated program variables that are either method parameter or it’s return value. The associated method is source/sink method.
- `getDeviceId()` is source method, return value is source variable
- `sendTextMessage(...,x,..)` is a sink method, where `x` is sink variable

Corresponds to relation of type `(srcComp : C, src : SRC, sinkComp : C, sink : SINK)`

**Definition 3.6:** A taint flow `(so, si)` represents a route from source to sink

**Definition 3.7:** `flow(p, so, q, si)` is true iff:
- `m` and `n` are source and sink methods for `so` and `si` respectively
- `calls(p,m)` and `call(q,n)` are true
- taint flow`(so,si)` exists
Predicate flows() : Example

```java
public class ListDevice extends Activity {
    protected void onCreate(Bundle bd) {
        Device n,m;
        ...
        String x = "deviceId=";
        String y = TelephonyManager.getDeviceId();
        String z = x.concat(y);
        m.f = z;
        n = m;
        String v = n.f;
        smsManager.sendTextMessage("3452",null,v,null,null,null);
    }
}

Figure 5: Example illustrating data flow
```

\textbf{flow(ListDevice, $getDeviceId$, ListDevice, !sendTextMessage) is True.}
Static Analysis

- Pointer analysis
- Data flow analysis for intents
- CCG construction
- Taint Analysis
Pointer Analysis

Notation for ‘x may point to y’: x \rightarrow y

Field-sensitive

Context-sensitive
- Call site sensitivity for static method calls
- Object sensitivity for virtual method calls

Anderson style
Data flow analysis for intents

Forward inter-procedural analysis

For each Intent variable $i$, the analysis tracks:
- $i_t \in$ Components
- $i_d \in$ Data types
- $i_a \in$ Actions

Values initialized to $\bot$

Join operator is the set union

Transfer function based on Android API

<table>
<thead>
<tr>
<th>Table 5: API for setting Intent attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Action</strong></td>
</tr>
<tr>
<td><strong>Data type</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Example: \texttt{x.setComponent(s)}

\textbf{Figure 6}: Transfer function for setComponent

If \(\Gamma(x_t)\) does not contain \(\bot\), \texttt{explicit}(x_t) \textbf{must} be true
Else \texttt{implicit}(x_t) \textbf{may} be true
ICCG Construction

Definition 4.1:
An ICCG for a program P is a graph \((N, E)\) such that:
- Nodes \(N\) are the set of components in \(P\)
- Edges \(E\) define a relation \(E \subseteq (N \times A \times D \times N)\) where
  - \(A\) and \(D\) are the domain of all actions and data types
ICCG Construction

icc_site(m,i) : Method m contains ICC site with intent i

P $\Rightarrow^* m : Component P transitively invokes m

intent_filter(P,A,D) : Component P has intent filter with action A and data D

• Extracted from the manifest.xml

\[
\begin{align*}
\text{icc_site}(m,i), \text{explicit}(i), \ P & \Rightarrow^* m \\
Q \in \Gamma(i_t), A \in \Gamma(i_a), D \in \Gamma(i_d) \\
(P, Q, A, D) & \in E
\end{align*}
\]  

(Explicit)

\[
\begin{align*}
\text{icc_site}(m,i), \text{implicit}(i), \ P & \Rightarrow^* m \\
A \in \Gamma(i_a), D \in \Gamma(i_d) \\
\text{intent_filter}(Q, A, D) \\
(P, Q, A, D) & \in E
\end{align*}
\]  

(Implicit)

Figure 8: ICCG construction rules
Taint Analysis

Annotations

• **Source**: for methods that read sensitive data (symbol: $)
• **Sink**: for methods that leak data outside the device (symbol: !)
• **Transfer**: for taint flow through android methods

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){ ... }
Taint Analysis Cont’d

New Predicate: tainted(o,l)
• Corresponds to relation of type (O : AbstractObj, L : SourceLabel)
• If true: any object represented by o may be tainted by l

m_i : i’th parameter of method m
• m_0 : ‘this’ variable
• m_{n+1} : return value (n is the number of parameters)

src(m_i,l) : i’th parameter of m is annotated as source label l
sink(m_i,l) : i’th parameter of m is passed to sink label l
transfer(m_i, m_j) : flow(m_i, m_j) is true
Taint Analysis Cont’d

\[
\text{src}(m_i, l), \ m_i \leftarrow o \\
\underline{\text{tainted}(o, l)}
\]

(Source)

\[
\text{tainted}(o_1, l), \ m_i \leftarrow o_1, \ m_j \leftarrow o_2 \\
\text{transfer}(m_i, m_j)
\]

(Transfer)

\[
\text{tainted}(o_2, l)
\]

\[
\text{tainted}(o, \text{so}), \ m_i \leftarrow o, \text{sink}(m_i, si) \\
\underline{\text{flow}(so, si)}
\]

(Sink)

Figure 10: Rules describing the taint analysis.
Performance Evaluation

Accuracy for known Malware 90%

• Performs poorly for BaseBridge (dynamic code loading)

11,215 Google apps scanned, only 16 reported malware

Approximately 350 seconds to analyze 27k lines of code

100% detection of obfuscated malware
Discussion

Taint Analysis vs Apposcopy
Maintaining malware database
Why Android? What generalizes to other systems?
What’s next?