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

y Feng et al [FSE ‘14]

resented by Maaz Ahmad

The Malware Problem

- (Feb, 2015) Motive Security Labs estimates 16 million infected mobile devices.[1]
- Nearly half of Android Malware attempt to steal personal data.
- Kaspersky Lab detected 29,695 new malware modifications in a quarter of a year.[2]

://www.alcatel-lucent.com/press/2015/alcatel-lucent-report-malware-2014-sees-rise-device-and-network-attacks-place-personal-and-workplace
://securelist.com/analysis/quarterly-malware-reports/37163/it-threat-evolution-q2-2013/

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

Vhat 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)

Signature Detection (ICCG)

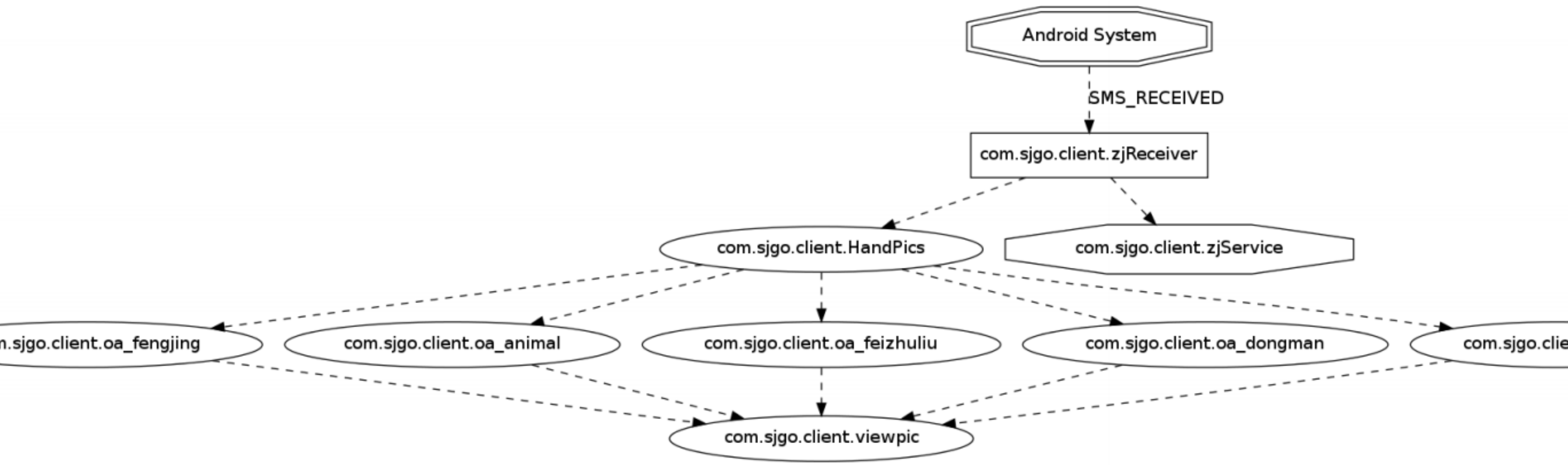
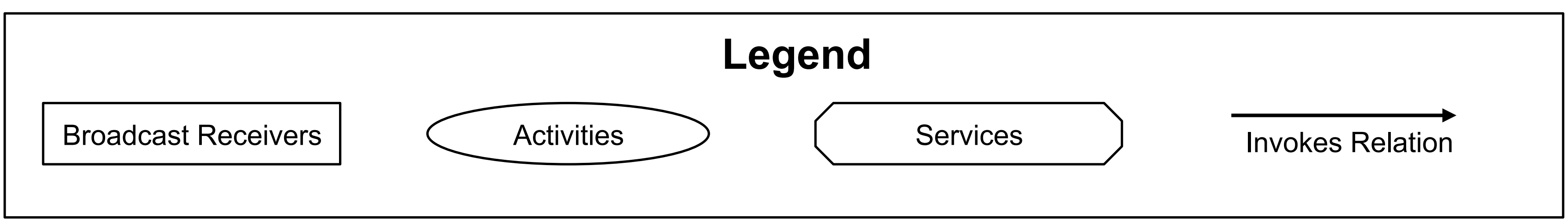


Figure 1: Partial ICCG for an instance of the GoldDream malware family



Signature Detection (Taint Analysis)

```
om.sjgo.client.zjService:
   $getSimSerialNumber -> !INTERNET
   $getDeviceId -> !INTERNET
   $getSubscriberId -> !INTERNET
   $getDeviceId -> !sendTextMessage
   $getSubscriberId -> !sendTextMessage
kboy.android.game.fiveInk.FiveLink:
   $ID -> !INTERNET
   $MODEL -> !INTERNET
et.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
  - *parent("Bill", "Mary")*
  - *GDEvent(SMS_RECEIVED)*
- A set of rules
  - *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)

CC Predicates

- nter-component communication predicates
- CC in Android revolves around Intents
- Methods that take Intent as parameter are called ICC methods
- nstructions that invoke ICC Methods are called ICC sites
- When ICC is initiated, life-cycle methods of the target component are nvoked

Table 1: A partial list of ICC-related APIs.

Activity	startActivity(Intent) startActivityForResult(Intent,int) startActivityIfNeeded(Intent,int) startNextMatchingActivity(Intent)
Service	startService(Intent) bindService(Intent)
oadcastReceiver	sendBroadcast(Intent) sendBroadcast(Intent,String) sendOrderedBroadcast(Intent,String)

Table 2: A partial list of life-cycle APIs.

Activity	onCreate(Bundle), onRestart(), onStart(), onResume(), onPause(), onStop(), onDestroy
Service	onCreate(), onBind(Intent), onStartCommand(Intent, int, i onDestroy()
BroadcastReceiver	onReceive(Context, Intent)

CC Predicates Cont'd

ntents passed to target may carry many types of information

Apposcopy only considers 'action' and 'data'

CC predicate represents inter-component communication in Android amework

- icc(s,t,a,d)
- Corresponds to relation of type (source : S, target : T, action : A, data : D)
- A and D may be $\perp$

CC Predicates Cont'd

efinition 3.1: Target of any ICC site is all components that receive passed
tent in some execution of the program.

efinition 3.2: m1 ➔ m2, if method m1 directly calls m2. m1 ➔* m2 if m1
ansitively calls m2.

efinition 3.3: The predicate icc(s,t,a,d) is true iff:

- m1 is a lifecycle method of s
- m1 ➔* m2
- m2 contains an icc site with target t
- The action and data values are a and d respectively

efinition 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 (component : C, callee : M)
- calls(c, m) is true iff:
  - n is a life-cycle method defined in component c
  - n ➔* m
- Help detect malware that abuse Android API methods

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

Operation & Description
<BroadcastReceiver: void abortBroadcast()> Block current broadcaster.
<Runtime: Process exec(java.lang.String)> Execute a command.
<System: void loadLibrary(java.lang.String)> Perform native call.
<PackageManager: List getInstalledPackages()> Get all application packages.
<DexClassLoader: void <init>(...,ClassLoader)> Load classes from .jar and .apk files.

Predicate flows()

Represents data flow to help detect sensitive information leak

Definition 3.5: Source and sink variables are annotated program variables that are
ther method parameter or it's return value. The associated method is source/sink
ethod.

- 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

```
blic 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);
  }
```

Figure 5: Example illustrating data flow

w(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 → 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 $\perp$
- Join operator is the set union
- Transfer function based on Android API

Table 5: API for setting Intent attrib

Target	`setComponent(ComponentName)`
	`setClassName(Context, Strin`
	`setClassName(String, String`
	`setClass(Context, Class)`

Action	`setAction(String)`
Data type	`setType(String), setData(UR` `setDataAndType(URI,String)`

Example: x.setComponent(s)

$$\frac{\text{must\_alias}(y, x)}{\Gamma \vdash \text{newval}(y, x, s) : [y_t \mapsto \{s\}]}$$

$$\frac{\text{may\_alias}(y, x), \quad \neg\text{must\_alias}(y, x)}{\Gamma \vdash \text{newval}(y, x, s) : [y_t \mapsto (\Gamma(y) \cup \{s\})]}$$

$$\frac{\neg\text{may\_alias}(y, x)}{\Gamma \vdash \text{newval}(y, x, s) : [y_t \mapsto \Gamma(y)]}$$

$$\frac{\Gamma \vdash \text{newval}(x_i, x, s) : \Gamma_i \quad (x_i \in \text{dom}(\Gamma))}{\Gamma \vdash \texttt{x.setComponent(s)} : \bigcup_i \Gamma_i}$$

Figure 6: Transfer function for setComponent

$$\frac{x \hookrightarrow o, \; y \hookrightarrow o}{\text{may\_alias}(x, y)} \qquad \frac{\begin{array}{c} x \hookrightarrow o, \neg\exists o'.x \hookrightarrow o' \\ y \hookrightarrow o, \neg\exists o'.y \hookrightarrow o' \\ |\gamma(o)| = 1 \end{array}}{\text{must\_alias}(x, y)} \qquad \frac{}{\text{must\_alias}(}$$

Figure 7: May and must aliasing relations

If $\Gamma(x_t)$ does not contain $\perp$, explicit(x_t) **must** be true

Else implicit(x_t) **may** be true

CCG Construction

efinition 4.1:

n ICCG for a program P is a graph (N, E) such that:

odes N are the set of components in P

dges E define a relation $E \subseteq (N \times A \times D \times N)$ where

and D are the domain of all actions and data types

CCG Construction

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

P ➔* m : Component P transitively invokes m

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

- Extracted from the manifest.xml

$$\frac{\begin{array}{c} icc\_site(m,i), explicit(i),\ \ P \rightsquigarrow\!\!\star\ m \\ Q \in \Gamma(i_t), A \in \Gamma(i_a), D \in \Gamma(i_d) \end{array}}{(P,Q,A,D) \in E} \quad \text{(Explicit)}$$

$$\frac{\begin{array}{c} icc\_site(m,i), implicit(i),\ \ P \rightsquigarrow\!\!\star\ m \\ A \in \Gamma(i_a), D \in \Gamma(i_d) \\ intent\_filter(Q,A,D) \end{array}}{(P,Q,A,D) \in E} \quad \text{(Implicit)}$$

Figure 8: ICCG construction rules

Taint Analysis

Annotations

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

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

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

//Transfer annotation in java.lang.String
@Flow(from="this",to="@return")
@Flow(from="s",to="@return")
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 tained 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

$$\frac{src(m_i, l),\ m_i \hookrightarrow o}{tainted(o, l)} \qquad \text{(Source)}$$

$$\frac{\begin{array}{c} tainted(o_1, l),\ m_i \hookrightarrow o_1,\ m_j \hookrightarrow o_2 \\ transfer(m_i, m_j) \end{array}}{tainted(o_2, l)} \qquad \text{(Transfer)}$$

$$\frac{tainted(o, so), m_i \hookrightarrow o, sink(m_i, si)}{flow(so, si)} \qquad \text{(Sink)}$$

gure 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?