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

Yu Feng
University of Texas at Austin, USA
yufeng@cs.utexas.edu

Saswat Anand
Stanford University, USA
saswat@cs.stanford.edu

Isil Dillig
University of Texas at Austin, USA
isil@cs.utexas.edu

Alex Aiken
Stanford University, USA
aiken@cs.stanford.edu

ABSTRACT

We present Apposcopy, a new semantics-based approach for identifying a prevalent class of Android malware that steals private user information. Apposcopy incorporates (i) a high-level language for specifying signatures that describe semantic characteristics of malware families and (ii) a static analysis for deciding if a given application matches a malware signature. The signature matching algorithm of Apposcopy uses a combination of static taint analysis and a new form of program representation called *Inter-Component Call Graph* to efficiently detect Android applications that have certain control- and data-flow properties. We have evaluated Apposcopy on a corpus of real-world Android applications and show that it can effectively and reliably pinpoint malicious applications that belong to certain malware families.

Categories and Subject Descriptors

D.4.6 [**Software Engineering**]: Security and Protection

General Terms

Security, Verification

Keywords

Android, Inter-component Call Graph, Taint Analysis

1. INTRODUCTION

As the most popular mobile operating system, the Android platform is a growing target for mobile malware [4]. Today, many of the malicious applications that afflict Android users exploit the private and monetized information stored in a user's smartphone. According to a recent report [3], nearly half of Android malware are multi-functional Trojans that steal personal data stored on the user's phone.

In response to the rapid dissemination of Android malware, there is a real need for tools that can automatically detect malicious applications that steal private user information. Two prevalent approaches for detecting such Android malware are *taint analyzers* and *signature-based detectors*:

Taint analyses, such as [17, 21], are capable of exposing applications that leak private user information. Unfortunately, since many benign apps also need access to sensitive data to perform their advertised functionality, not every app that leaks user information can be classified as malware. For instance, an email client application will necessarily "leak" email addresses of the user's contacts in order to perform its functionality. Thus, taint analyses cannot automatically distinguish benign apps from malware, and a security auditor must invest significant effort in order to determine whether a given information flow constitutes malicious behavior.

Signature-based malware detectors, including commercial virus scanners, classify a program as malware if it contains a sequence of instructions that is matched by a regular expression. As shown in a recent study, malware detectors that are based on syntactic low-level signatures can be easily circumvented using simple program obfuscations [34]. Hence, these malware signatures must be frequently updated as new variants of the same malware family emerge.

In this paper, we present Apposcopy, a new semantics-based approach for detecting Android malware that steal private user information. Drawing insights from the respective advantages of pattern-based malware detectors and taint analyzers, Apposcopy incorporates (i) a high-level specification language for describing semantic characteristics of Android malware families, and (ii) a powerful static analysis for deciding if a given application matches the signature of a malware family. The semantic, high-level nature of the signature specification language allows analysts to specify key characteristics of malware families without relying on the occurrence of specific instruction or byte sequences, making Apposcopy more resistant to low-level code transformations.

The signature-based specification language provided by Apposcopy allows specifying two types of semantic properties —control-flow and data-flow— of Android applications. An example of a control-flow property is that the malware contains a broadcast receiver which launches a service upon the completion of some system event. An example of a data flow property is that the malware reads some private data

[*]This work was sponsored by the Air Force Research Laboratory, under agreement number FA8750-12-2-0020

FSE '14, November 16–22, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...$15.00.

of the device and sends it through a designated sink(e.g., Internet, SMS message, etc.).

To match the signatures specified in this language, Apposcopy's static analysis relies on two key ingredients. First, we construct a new high-level representation of Android applications called the *inter-component callgraph (ICCG)*, which is used to decide whether an Android application matches the control flow properties specified in the signature. Second, Apposcopy incorporates a static taint analysis which is used for deciding whether a given application matches a specified data-flow property.

We have evaluated Apposcopy on a corpus of real-world Android applications and show that it can effectively and reliably pinpoint malicious applications –including obfuscated ones– that belong to certain malware families. Despite the theoretical undecidability of the semantic signature matching problem, Apposcopy yields both few false positives and few false negatives on current Android applications.

2. MOTIVATING EXAMPLE

In this section, we illustrate our approach using a simplified version of the GoldDream malware family. As described in Jiang's security report [27], the key characteristic of members of this family is that they register a receiver for certain system events such as SMS messages or outgoing phone calls. When these events trigger the execution of code in the receiver, the malware then starts a background service for sending private user information, such as the phone's unique IMEI number and subscriber id, to a remote server.

2.1 GoldDream Signature in Apposcopy

To detect a sample of GoldDream malware, an analyst first writes a signature of this malware family in our Datalog-based language. In this case, the behavior of GoldDream is captured by the specification in Figure 2. Here, lines 1-2 introduce a new user-defined predicate `GDEvent(x)` which describes the events that the `GoldDream` malware listens for. In this case, `GDEvent(x)` evaluates to true when `x` is either `SMS_RECEIVED` or `NEW_OUTGOING_CALL` but to false otherwise.

Using this predicate, lines 3-7 describe the signature of the GoldDream malware family. In this case, the signature uses three kinds of predicates provided by Apposcopy:

Component type predicates, such as `receiver(r)` and `service(s)`, specify that `r` and `s` are `BroadcastReceiver` and `Service` components in the Android framework.

Control-flow predicates: An example of a control flow predicate is `icc`, which describes inter-component communication in Android. In our example, `icc(SYSTEM, r, e, _)` expresses that the Android system invokes component `r` when system event `e` happens, and `icc*(r, s)` means that component `r` transitively invokes component `s`.

Data-flow predicates, such as `flow(x, so, y, si)`, express that the application contains a flow from source `so` in component `x` to a sink `si` in component `y`. Hence, lines 6-7 state that component `s` sends the device and subscriber id of the phone over the Internet.

Therefore, according to the signature(simplified version) in Figure 2, an application A belongs to the GoldDream malware family if (i) A contains a broadcast receiver that listens for system events `SMS_RECEIVED` or `NEW_OUTGOING_CALL` (lines 3, 4), and (ii) this broadcast receiver starts a service which then leaks the device id and subscriber id over the Internet (lines 5-7).

```
1. GDEvent(SMS_RECEIVED).
2. GDEvent(NEW_OUTGOING_CALL).
3. GoldDream :-  receiver(r),
4.               icc(SYSTEM, r, e, _), GDEvent(e),
5.               service(s), icc*(r, s),
6.               flow(s, DeviceId, s, Internet),
7.               flow(s, SubscriberId, s, Internet).
```

Figure 2: GoldDream signature (simplified)

2.2 GoldDream Malware Detection

Given an Android application A and malware signature S, Apposcopy performs static analysis to decide if app A matches signature S. Apposcopy's static analysis has two important ingredients: (i) *construction of the ICCG*, which determines the truth values of control-flow predicates used in the signature, and (ii) *static taint analysis*, which is used to decide the truth values of data-flow predicates.

Figure 1 shows a partial ICCG for an instance of the GoldDream malware family. Nodes in the ICCG correspond to components, and node labels denote component names. The shapes of the nodes indicate component types: Rectangles denote broadcast receivers, ellipses indicate activities, and polygons are services. An ICCG edge from one node A to another node B means that component A starts component B, for example, by calling the `startActivity` method of the Android SDK. The edges in the ICCG may also be labeled with additional information, such as system events.

Going back to the specification from Section 2.1, the ICCG shown in Figure 1 matches the sub-query

$$\mathtt{receiver(r), icc(SYSTEM, r, e, \_), GDEvent(e),}$$
$$\mathtt{service(s), icc * (r, s)}$$

because (i) there exists a node in the ICCG representing a receiver component (namely, `zjReceiver`), (ii) there is an edge from the node representing the Android system to `zjReceiver` labeled with `SMS_RECEIVED`, (iii) `zjReceiver` has an outgoing edge to a service component called `zjService`.

Next, to decide data-flow queries, Apposcopy performs taint analysis tracking flows from *sources* to *sinks*. Here, sources represent sensitive data, such as the phone's device id, and sinks represent operations that may leak data, such as sending text messages. For our example application, the taint analysis yields the following result:

```
com.sjgo.client.zjService:
    $getSimSerialNumber -> !INTERNET
    $getDeviceId -> !INTERNET
    $getSubscriberId -> !INTERNET
    $getDeviceId -> !sendTextMessage
    $getSubscriberId -> !sendTextMessage
cxboy.android.game.fiveInk.FiveLink:
    $ID -> !INTERNET
    $MODEL -> !INTERNET
net.youmi.android.AdActivity:
    $getDeviceId -> !WebView
    $ExternalStorage -> !WebView
```

Here, Apposcopy has identified source-sink flows in three components, `zjService`, `FiveLink`, and `AdActivity`. For example, the first three lines under `zjService` indicate that it sends the phone's serial number, device id, and subscriber id over the Internet. Recall that the GoldDream malware signature includes the data-flow query:

```
flow(s, DeviceId, s, Internet),
flow(s, SubscriberId, s, Internet)
```

where `s` is a service. Since the taint analysis reveals that `zjService` leaks the device and subscriber id to the Internet, this query evaluates to true, and Apposcopy concludes this application is GoldDream malware. Observe that, although

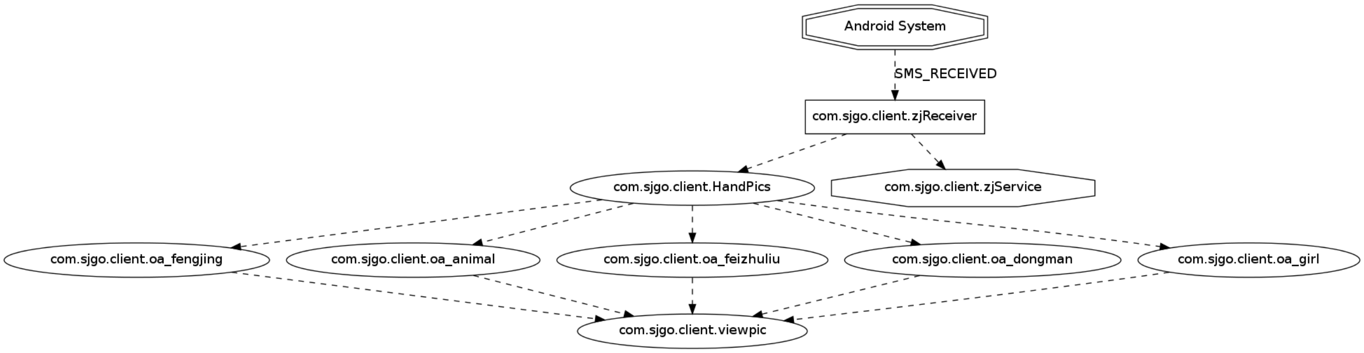
Android System
SMS_RECEIVED
com.sjgo.client.zjReceiver
com.sjgo.client.HandPics
com.sjgo.client.zjService
com.sjgo.client.oa_fengjing
com.sjgo.client.oa_animal
com.sjgo.client.oa_feizhuliu
com.sjgo.client.oa_dongman
com.sjgo.client.oa_girl
com.sjgo.client.viewpic

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

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)`

BroadcastReceiver	`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, int)`, `onDestroy()`
BroadcastReceiver	`onReceive(Context, Intent)`

there are other source-sink flows in this example (such as from `DeviceId` to `WebView` in `AdActivity`), these other flows do not necessarily correspond to malicious behavior.

3. MALWARE SPEC LANGUAGE

This section describes Apposcopy's malware signature language, which is a Datalog program augmented with built-in predicates. For each malware family, the user defines a unique predicate that serves as the signature for this malware family. The user may also define additional helper predicates used in the signature. In what follows, we first give some background on Datalog, and then describe the the syntax and semantics of Apposcopy's built-in predicates.

3.1 Datalog Preliminaries

A Datalog program consists of a set of *rules* and a set of *facts*. Facts simply declare predicates that evaluate to true. For example, `parent("Bill", "Mary")` states that Bill is a parent of Mary. Each Datalog rule is a Horn clause defining a predicate as a conjunction of other predicates. For example, the rule:

```
ancestor(x, y) :- parent(x, z), ancestor(z, y).
```

says that `ancestor(x, y)` is true if both `parent(x, z)` and `ancestor(z, y)` are true. In addition to variables, predicates can also contain constants, which are surrounded by double quotes, or "don't cares", denoted by underscores.

Datalog predicates naturally represent relations. Specifically, if tuple (x, y, z) is in relation A, this means the predicate $A(x, y, z)$ is true. In what follows, we write the type of a relation $R \subseteq X \times Y \times \ldots$ as $(s_1 : X, s_2 : Y, \ldots)$, where $s1$, $s2$, ... are descriptive texts for the corresponding domains.

3.2 Apposcopy's Built-in Predicates

We now describe the syntax and semantics of the four classes of built-in predicates provided by Apposcopy.

3.2.1 Component Type Predicates

Component type predicates in Apposcopy represent the different kinds of components provided by the Android framework. An Android application consists of four kinds of components, *Activity*, *Service*, *BroadcastReceiver*, and *ContentProvider*. Activity components form the basis of the user interface, and each window of the application is typically controlled by an activity. Service components run in the background and remain active even if windows are switched. BroadcastReceiver components react asynchronously to messages from other applications. Finally, ContentProvider components store data relevant to the application, usually in a database, and allow sharing data across applications.

Corresponding to each of these Android components, Apposcopy provides pre-defined predicates called `service(C)`, `activity(C)`, `receiver(C)`, and `contentprovider(C)` which represent the type of component `C`. For example, `activity(C)` is true if `C` is an `Activity`. Each of these four predicates correspond to relations of type $(comp : C)$ where domain C is the set of all components in the application.

3.2.2 Predicate `icc`

A key idea behind inter-component communication (ICC) in Android is that of Intents, which are messages passed between components. Intents are used to start Activities; start, stop, and bind Services; and broadcast information to Broadcast Receivers. Table 1 shows a list of Android APIs that are used in ICC. We refer to those methods, all of which take Intent objects as arguments, as *ICC methods*. We use the term *ICC site* to represent a statement that invokes one of the ICC methods listed in Table 1. When component `a` initiates ICC with component `b`, the Android system will eventually call *life-cycle* methods associated with component `b`. The life-cycle methods are shown in Table 2.

While an Intent object passed to ICC methods can contain many different types of information, Apposcopy's signature language takes into account only two kinds: action and data. *Action* is a string that represents the type of action that the receiving component needs to perform, and *data* specifies the type of data that the component needs to operate on. For example, a component for editing images will receive intents with corresponding action `EDIT` and data type `image/*`.

Apposcopy's `icc` predicate represents inter-component communication in Android and corresponds to a relation of type $(source : C, target : C, action : A, data : D)$ where domain

```
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.

```
<activity android:name="MsgAct">
  <intent-filter
    <action name="android.intent.action.SEND" />
    <data mimeType="text/plain" />
 </intent-filter>
</activity>
```

Figure 4: A snippet of AndroidManifest.xml

C is the set of all components in the application, and A and D are the sets of all Action and Data values defined by the Android system. Since not all intents are required to have action and data values, domains A and D also include a special element $\perp$, which indicates no value. Intuitively, if predicate `icc(p,q,a,d)` is true, this means that component `p` passes an intent to `q` through invocation of an ICC method, and `a` and `d` represent the action and data strings of the intent. To formally state the semantics of the `icc` predicate, we first define targets of ICC sites:

Definition 3.1. *The* target of an ICC site*, m(i,...), is the set of components that will receive the intent stored in variable* `i` *in some execution of the application.*

Definition 3.2. *We write $m_1 \rightsquigarrow m_2$ iff method m_1 directly calls m_2 in some execution of the program. We define $\rightsquigarrow\star$ to be the reflexive transitive closure of $\rightsquigarrow$.*

In other words, $m_1 \rightsquigarrow\star m_2$ if m_1 transitively calls m_2. We now define the semantics of the `icc` predicate as follows:

Definition 3.3. *The predicate* `icc(p,q,a,d)` *is true iff (i) m_1 is a life-cycle method defined in component* `p`*, (ii) $m_1 \rightsquigarrow\star m_2$, (iii) m_2 contains an ICC site whose target is component* `q`*, and (iv) the action and data values of the intent are* `a` *and* `d`*, respectively.*

Example 1. *Consider the code in Figure 3, which defines two activities,* `MainAct` *and* `MsgAct`*. The* `onCreate` *life-cycle method of* `MainAct` *calls* `foo` *which initiates ICC at line 10 by calling* `startActivity` *with intent* `i`*. The action associated with intent* `i` *is* `SEND` *and the data type is* `text/plain`*. According to the manifest in Figure 4,* `MsgAct` *declares an* intent filter *for* `SEND` *actions on data type* `text/plain`*, meaning that* `msgAct` *is a target of the ICC site at line 10. Hence, predicate* `icc(MainAct, MsgAct, SEND, text/plain)` *is true.*

We now define the predicate `icc*(p, q)`, which is true if component `p` can transitively launch component `q`:

Definition 3.4. *Let* `icc_direct(p,q)` *be a binary predicate which is true iff* `icc(p, q, _, _)` *is true. The predicate* `icc*` *is the reflexive transitive closure of* `icc_direct`*.*

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.

Our malware signature language includes the predicate `icc*` because it allows writing signatures that are resilient to high-level control flow obfuscation. In particular, if the signature contains the predicate `icc*(p, q)`, adding or deleting dummy components for the purposes of detection evasion will not affect the truth value of this predicate.

3.2.3 Predicate `calls`

Apposcopy provides another control-flow predicate, called `calls`, representing a relation of type $(comp : C, callee : M)$ where domains C and M represent the set of all components and methods in the program respectively. Intuitively, `calls(c,m)` is true if method `m` is called by component `c`. More precisely, `calls(c, m)` is true iff `n` is a life-cycle method defined in component `c` and `n` $\rightsquigarrow\star$ `m`.

The `calls` predicate is useful for defining malware signatures because it can be used to check if a component calls Android API methods that can be abused by malware. Table 3.2.4 lists a few of such dangerous methods.

3.2.4 Predicate `flows`

Apposcopy provides a data-flow predicate `flows` used for querying whether an app leaks sensitive data. More specifically, taint flow is defined in terms of sources and sinks.

Definition 3.5. *A* source (resp. sink) *is a labeled (i.e., annotated) program variable that is either a method parameter or method return value. The corresponding method is referred to as the* source method (resp. sink method).

An example of a source is the return value of method `getDeviceId`, which yields the phone's unique device id. An example of a sink is the third parameter of `sendTextMessage`, which corresponds to the text to be sent through SMS. Source and sink annotations are discussed in Section 4.3.1, and Table 3.2.4 shows a partial list of source and sink methods.

The `flow` predicate represents a relation of type $(srcComp : C, src : \mathrm{SRC}, sinkComp : C, sink : \mathrm{SINK})$ where domain C is the set of components, and SRC and SINK are the sets of all sources and sinks in the program. To define the semantics of the `flow` predicate, we first define *taint flow*:

Definition 3.6. *A* taint flow (so, si) *represents that a source labeled so can flow to a sink labeled si through a series of assignments or matching loads and stores.*

Definition 3.7. *The predicate* `flow(p,so,q,si)` *is true iff (i)* `m` *and* `n` *are the source and sink methods corresponding to source* `so` *and sink* `si`*, (ii)* `calls(p, m)` *and* `calls(q,n)` *are both true, and (iii) there exists a taint flow* `(so,si)`*.*

Example 2. *Consider the code in Figure 5, where the return value of* `getDeviceId` *is a source labeled* `$getDeviceId`*,*

Table 4: Examples of APIs with source and sink annotations

Sources	Sinks
`<TelephonyManager: String getDeviceId()>`	`<SmsManager: void sendTextMessage(...)>`
`<TelephonyManager: String getSimSerialNumber()>`	`<SmsManager: void sendMultiparTextMessage(...)>`
`<TelephoneyManager: String getSubscriberId()>`	`<SmsManager: void sendDataMessage(...)>`
`<TelephoneyManager: String getVoiceMailNumber()>`	`<DataOutputStream: write(...)>`
`<TelephoneyManager: String getSimOperator()>`	`<DatagramSocket: void send(...)>`
`<Location: double getLatitude()>`	`<AbstractHttpClient: HttpResponse execute(...)>`
`<Location: double getLongitude()>`	`<Ndef: void writeNdefMessage(...)>`

```
public class ListDevice extends Activity {
    protected void onCreate(Bundle bd) {
1.      Device n,m;
2.        ...
3.      String x = "deviceId=";
4.      String y = TelephonyManager.getDeviceId();
5.      String z = x.concat(y);
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

and the third parameter of `sendTextMessage` *is a sink labeled* `!sendTextMessage`*. This application exhibits a taint flow from* `$getDeviceId` *to* `!sendTextMessage` *because variable* `y` *holding the return value of* `getDeviceId` *can flow to variable* `v` *due to the chain of assignments, loads, and stores performed in lines 5-8. Hence, the following predicate evaluates to true:*

`flow(ListDevice,$getDeviceId,ListDevice,!sendTextMessage)`

4. STATIC ANALYSES

This section describes Apposcopy's static analyses for deciding whether an application matches a malware signature. The main idea is to compute an over-approximation of the `icc`, `calls`, and `flow` relations.

4.1 Pointer Analysis & Callgraph Contruction

In order to build the inter-component callgraph and track information flow, Apposcopy starts by performing a pointer analysis, which computes the set of abstract heap objects that each variable may point to. In the remainder of the paper, we use the notation $v \hookrightarrow o$ to denote that variable v may point to an abstract heap object represented by o in some execution of the program.

Since the precision of the underlying pointer analysis is critical for detecting malware with few false alarms, we use a field- and context-sensitive Andersen-style pointer analysis [7]. For context-sensitivity, we use a hybrid approach that combines call-site sensitivity [30] and object-sensitivity [31]. In particular, our approach is similar to the technique described in [28] and uses call-site sensitivity for static method calls and object-sensitivity for virtual method calls.

Another key ingredient of our malware detection algorithm is callgraph construction, which is used for resolving the targets of virtual method calls. Since callgraph precision has significant impact on the precision of the ICCG, Apposcopy computes the callgraph on-the-fly, simultaneously refining the targets of virtual method calls and points-to sets until a fixed point is reached. The set of edges in the resulting callgraph represent the relation $\rightsquigarrow$ from Definition 3.2. An edge in the callgraph from method m_1 to method m_2 corresponds to $m_1 \rightsquigarrow m_2$. Similarly, $m_1 \rightsquigarrow^{\star} m_2$ represents that there exists a path in the callgraph from m_1 to m_2.

Table 5: API for setting Intent attributes

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

Action	`setAction(String)`

Data type	`setType(String), setData(URI),`
	`setDataAndType(URI,String)`

4.2 Inter-component Control-flow Analysis

We now describe the construction of the inter-component call graph (ICCG), which is used for deciding ICC queries.

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, and 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 defined by the Android system augmented with the element $\perp$.*

In other words, ICCG is a directed graph where nodes are components in an application, and edges are labeled with actions and data types. The special element $\perp$ indicates that the intent defining the ICC does not have action or data type information associated with it. Given a callgraph and the results of the pointer analysis, Apposcopy constructs the ICCG by performing two key steps that we explain next.

4.2.1 Data Flow Analysis for Intents

The first step of ICCG construction is a data flow analysis to track the information stored in Intent objects. Specifically, we track three kinds of information about intents:

- **Target:** In the Android framework, a component can specify the target of an Intent (and, hence, the ICC) by calling the methods shown in the first row of Table 5. Such intents whose targets have been explicitly specified are called *explicit intents.*

- **Action:** A component can specify the action that the ICC target needs to perform by calling the methods shown in the second row of Table 5.

- **Data type:** An application can specify the data type that the recipient component needs to operate on by calling the methods shown in the last row of Table 5.

When a component does not specify the target of an intent explicitly, the Android system resolves the recipient components of the ICC based on *intent filters* declared in the manifest file. An intent filter specifies the kinds of actions that a component will respond to. For instance, consider the `AndroidManifest.xml` file from Figure 4. Here, `MsgAct` declares that it responds to intents whose corresponding action and data type are `SEND` and `text/plain` respectively.

In order to build the ICCG, Apposcopy first performs a forward interprocedural dataflow analysis, called the *intent analysis*, which overapproximates the target, action, and

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

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

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

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

Figure 6: Transfer function for `setComponent`

data type associated with each intent object. Specifically, for each variable `i` of type `Intent`, our analysis tracks three variables i_t, i_a, and i_d whose domains are sets of components, actions, and data types respectively. We initialize the dataflow value for each variable to be $\{\bot\}$ and define the join operator as set union.

Table 5 shows the set of Android API methods for setting attributes of Intent objects. Since other methods do not change attributes of intents, the transfer function for any statement not included in Table 5 is the identity function.

Figure 6 shows the transfer function for `setComponent` in the form of inference rules. Since transformers for the other statements from Table 5 are very similar, we only focus on `setComponent` as a representative. In Figure 6, environment Γ denotes data flow facts for targets of intent variables as a mapping from each variable to its corresponding dataflow value. Now, consider the statement `x.setComponent(s)` where `x` is a variable of type Intent and `s` is a string specifying the target component. If another variable `y` is an alias of `x`, then the target of `y` will also be affected by this statement. Hence, our transfer function must update the data flow values of all variables that are aliases of `x`. Now, there are three cases to consider. If `x` and `y` are guaranteed to be aliases, as in the first rule of Figure 6, then we can kill its old value and update its new value to $\{s\}$ (i.e., a strong update). In the second rule, if `y` *may* alias `x`, then `y`'s target could either remain unchanged or become `s`; hence, we only perform a weak update. Finally, if `y` and `x` do not alias, then `y`'s target definitely remains unchanged and its existing dataflow values are preserved. The *may_alias* and *must_alias* relations used in Figure 6 are defined according to Figure 7. In the definition of the *must_alias* relation, $\gamma(o)$ represents the set of concrete memory locations represented by abstract memory location `o`.

Based on the results of this data flow analysis, we can now determine whether an intent is explicit or implicit. Specifically, if $\Gamma(x_t)$ does not contain $\bot$, the target of the intent must have been explicitly specified. Hence, we write *explicit*(x) if $\bot \notin \Gamma(x_t)$. Otherwise, `x` *may* be an implicit intent, denoted *implicit*(x).

Example 3. *Consider again the code from Figure 3. Here, for Intent* `i` *declared at line 7, we have* $\Gamma(i_t) = \{\bot\}$, $\Gamma(i_a) = \{\texttt{action.SEND}\}$, $\Gamma(i_d) = \{\texttt{text/plain}\}$ *For the intent* `n` *at line 13, our analysis computes* $\Gamma(n_t) = \{\texttt{MsgAct}\}$, $\Gamma(n_a) = \{\bot\}$, $\Gamma(n_d) = \{\bot\}$. *Since* $\bot \notin \Gamma(n_t)$, *we conclude* explicit(n). *On the other hand,* `i` *is identified to be implicit.*

4.2.2 ICCG Construction

We now describe ICCG construction using the results of the intent analysis. In what follows, we write *icc_site*(m, i) to denote that method m contains an ICC site with intent i, and we write $P \rightsquigarrow^{\star} m$ to indicate that component P has a life-cycle method m' and $m' \rightsquigarrow^{\star} m$ Finally, the predicate *intent_filter*(P, A, D) means that component P declares an intent filter with action A and data type D. This information is extracted from the application's manifest file.

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

Figure 7: May and must aliasing relations

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

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

Figure 8: ICCG construction rules

Figure 8 shows the ICCG construction rules. The first rule (Explicit) considers ICC sites in method m where intent i has its target component explicitly specified (i.e., $Q \in \Gamma(i_t)$ and $Q \neq \bot$). In this case, if method m is reachable from component P, we add an edge from component P to Q in the ICCG. Furthermore, if $A \in \Gamma(i_a)$ and $D \in \Gamma(i_d)$, the edge from P to Q has action label A and data type label D.

The second rule in Figure 8 applies to ICC sites where intent i may be implicit. If $A \in \Gamma(i_a)$ and $D \in \Gamma(i_d)$, we need to determine all components Q that declare intent filters with action A and data type D. Hence, we add an edge from component P to Q if *intent_filter*(Q, A, D) is true.

Example 4. *Consider again the code from Figure 3 and the data-flow facts* Γ *computed in Example 3. Using the Implicit rule for the ICC site in method* `foo` *with intent* `i`*, we infer the edge* $(\texttt{MainAct}, \texttt{MsgAct}, \texttt{action.SEND}, \texttt{text/plain})$. *Using the Explicit rule for ICC site in method* `bar` *with intent* `n`*, we add the edge* $(\texttt{MainAct}, \texttt{MsgAct}, \bot, \bot)$.

4.3 Taint Analysis

We now describe Apposcopy's taint analysis for answering data-flow queries. We first discuss how we annotate sources and sinks and then describe our taint propagation rules.

4.3.1 Annotations

Apposcopy provides three types of annotations called *source*, *sink*, and *transfer* annotations. Source annotations are used to mark Android framework methods that read sensitive data, and sink annotations indicate methods that leak data outside of the device. In contrast, transfer annotations are used for describing taint flow through Android SDK methods. In our approach, transfer annotations are necessary because Apposcopy analyzes the source code of Android applications, but *not* the underlying implementation of the Android platform. Hence, we have manually written annotations for Android SDK methods that propagate taint between parameters and return values.

Example 5. *Figure 9 shows representative examples of source, sink, and transfer annotations. All three kinds of specifications are written using the* `@Flow` *annotation, and sources and sinks are indicated by the special prefixes* `$` *and* `!` *respectively. In Figure 9, the annotation at line 2 is a source annotation, indicating that* `getDeviceId`*'s return value is a taint source. The sink annotation at line 8 indicates that*

```
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){ ... }
```

Figure 9: Source, Sink and Transfer annotations.

`sendTextMessage` *is a sink for its formal parameter* `text`*. Lines 11-12 correspond to transfer annotations for the Android library method* `concat` *which is not analyzed by Apposcopy. According to line 11, if the* `this` *parameter of* `concat` *is tainted, then its return value is also tainted. Similarly, the annotation at line 12 states that if parameter* `s` *of* `concat` *is tainted, then so is the return value.*

We emphasize that the source, sink, and transfer annotations are *not* written by individual users of Apposcopy, which already comes with a set of built-in annotations that we have written. Apposcopy uses the same pre-defined annotations for analyzing every Android application.

4.3.2 Static Taint Analysis

We describe Apposcopy's taint analysis using the inference rules shown in Figure 10, which define two predicates $\mathit{tainted}(o, l)$ and $\mathit{flow}(so, si)$. The predicate *tainted* represents a relation (O : AbstractObj, L : SourceLabel) where domain O is the set of all abstract heap objects and L is the set of all source labels, such as `$getDeviceId`. If $\mathit{tainted}(o, l)$ is true, this means that any concrete heap objected represented by o may be tainted with l. The predicate $\mathit{flow}(so, si)$ represents a relation of type (L : SourceLabel, L : SinkLabel). If $\mathit{flow}(so, si)$ is true, this means that source so may reach sink si. Hence, the *flow* predicate is a static over-approximation of the taint flow relation introduced in Definition 3.6.

All of the rules in Figure 10 use the notation m_i to denote the i'th parameter of method m. For uniformity of presentation, we represent the `this` pointer as m_0 and the return value as m_{n+1} where n is the number of arguments of m.

The first rule labeled Source in Figure 10 describes taint introduction. In this rule, we use the notation $\mathit{src}(m_i, l)$ to denote that the i'th parameter of m is annotated with source label l as described in Section 4.3.1. Hence, if variable m_i may point to heap object o and m_i is annotated as source l, then heap object o also becomes tainted with label l.

The second rule called Transfer performs taint propagation. Here, the predicate $\mathit{transfer}(m_i, m_j)$ corresponds to the transfer annotations from Section 4.3.1 and indicates there is a flow from the i'th to the j'th parameter of Android SDK method m. According to this rule, if (i) m_i can flow to m_j, (ii) m_i and m_j may point to heap objects o_1 and o_2 respectively, and (iii) o_1 is tainted with label l, then o_2 also becomes tainted with label l.

The third rule labeled Sink defines the *flow* predicate using the *tainted* predicate. Here, the notation $\mathit{sink}(m_i, si)$ means that the i'th parameter of m_i is passed to some sink labeled si. Hence, according to this rule, if m_i is passed to sink si, and m_i may point to a heap object o that is tainted with label so, then there may be a flow from source so to sink si.

The taint analysis of Apposcopy consists of applying the rules from Figure 10 until a fixed-point is reached. Observe that the rules from Figure 10 do not describe transformers for individual instructions such as stores because we use the points-to facts computed by a whole-program pointer analysis. That is, if any variable v in the program may flow to m_i through a chain of heap reads and writes, we will have $m_i \hookrightarrow o_v$ where o_v is the location pointed to by v. [1]

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

$$\frac{\mathit{tainted}(o_1, l),\ m_i \hookrightarrow o_1,\ m_j \hookrightarrow o_2 \quad \mathit{transfer}(m_i, m_j)}{\mathit{tainted}(o_2, l)} \qquad \text{(Transfer)}$$

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

Figure 10: Rules describing the taint analysis.

Example 6. *Consider the code in Figure 5 and annotations in Figure 9. Say* `x`, `y`, `z` *point to heap objects* o_1, o_2, o_3 *respectively. Since* $\texttt{src}(\texttt{getDeviceId}_{\text{return}}, \texttt{\$getDeviceId})$ *and* $y \hookrightarrow o_2$*, the Source rule infers* $\texttt{tainted}(o_2, \texttt{\$getDeviceId})$. $\texttt{transfer}(\texttt{concat}_1, \texttt{concat}_{\text{return}})$ *denotes a transfer annotation at line 5 where* $\texttt{concat}_1 \hookrightarrow o_2$, $\texttt{concat}_{\text{return}} \hookrightarrow o_3$*; thus, the Transfer rule infers* $\texttt{tainted}(o_3, \texttt{\$getDeviceId})$*. Finally, consider the call to method* `sendTextMessage` *which has a sink annotation* $\texttt{sink}(\texttt{sendTextMessage}_3, \texttt{!getDeviceId})$*. Since the argument* `v` *and* `z` *are aliases, we have* $\texttt{sendTextMessage}_3 \hookrightarrow o_3$*. Hence, we deduce* $\texttt{flow}(\texttt{\$getDeviceId}, \texttt{!sendTextMessage})$.

5. IMPLEMENTATION & EVALUATION

Our implementation consists of about 30,000 lines of Java and uses several publicly-available software such as Soot [35] and bddbddb [36]. Soot is used to convert Android's .apk files to *Jimple*, a higher-level intermediate representation. A pre-processing step processes Jimple instructions to extract various types of program facts, and our static analyses are specified as Datalog programs. The bddbddb system takes as input the Datalog specification of a static analysis and extracted program facts and outputs the results of the analysis. Apposcopy's static analyses use manually-written models of the Android framework; currently, we have models for about 1,100 classes that are relevant for our analyses.

To evaluate the effectiveness and accuracy of Apposcopy, we performed four sets of experiments, including evaluation on (i) known malware, (ii) Google Play apps, (iii) obfuscated malware. In addition, (iv) we also compare Apposcopy with another research tool called Kirin for Android malware detection. The remainder of this section describes the details and results of our evaluations.

5.1 Evaluation on Known Malware

In our first experiment, we evaluate the effectiveness of Apposcopy on 1027 malware instances from the Android Malware Genome project [1], which contains real malware collected from various sources, including Chinese and Russian third-party app markets. All of these malicious applications belong to known malware families, such as DroidKungFu, Geinimi, and GoldDream. To perform this experiment, we manually wrote specifications for the malware families included in the Android Malware Genome Project. For this purpose, we first read the relevant reports where available and inspected a small number (1-5) of instances

[1] For simplicity of presentation, we assume every variable is a pointer to a heap object.

Table 6: Examples of Apposcopy's signatures.

Malware family	Signature
ADRD	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).
BeanBot	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).
CoinPirate	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).

for each malware family. Table 6 shows signatures that we wrote for some of these malware families.

Table 7 presents the results of this experiment. The first column indicates the malware family, and the second column shows the number of analyzed instances of that malware family. The next two columns show the number of false negatives (FN) and false positives (FP) respectively. In this context, a false negative arises if an application A belongs to a certain malware family F but Apposcopy cannot detect that A is an instance of F. Conversely, a false positive arises if an application A does not belong to malware family F but Apposcopy erroneously reports that it does. The final column of Table 7 reports Apposcopy's overall accuracy, which is calculated as the number of correctly classified instances divided by the total number of samples.

As shown in the last row of Table 7, the overall accuracy of Apposcopy over all malware instances that we analyzed is 90.0%. That is, it can correctly classify approximately 9 out of 10 malware instances accurately.

However, looking at the results more closely, we see that Apposcopy performs quite poorly on the BaseBridge malware family. Specifically, among the 121 samples that we analyzed, Apposcopy only classifies 46 of these applications as instances of BaseBridge. Upon further inspection of this family, we found that many of the BaseBridge instances dynamically load the code that performs malicious functionality. Such behavior can inherently not be detected using static analysis and causes Apposcopy to yield many false negatives. Observe that, if we exclude BaseBridge from our samples, the overall accuracy of Apposcopy rises to 96.9%.

For the other malware families for which Apposcopy yields false negatives, there are several contributing factors. First, since we have written the specifications by inspecting only a small number of samples, our signatures may not adequately capture the essential characteristics of *all* instances of that family. Second, the malware family may have some key feature that is not expressible in our malware specification language. For example, if a given malware performs malicious functionality without leaking sensitive data, Apposcopy will be unable to detect it. A third contributing factor for false negatives is due to missing models. Specifically, while our static analysis is sound, we do not analyze the underlying code of the Android system, but instead rely on *method stubs* that capture the relevant behavior of the Android SDK. If these applications call SDK methods for which we have not provided stubs, Apposcopy may yield false negatives.

Based on the data from Table 7, we observe that Apposcopy reports very few false positives. Among the 1027 malware samples, Apposcopy reports two instances of the Geinimi family as instances of both Geinimi as well as the DroidKungFu family. This corresponds to an overall false positive ratio of less than 0.2%, indicating that Apposcopy's static analysis is precise enough to accurately answer control- and data-flow queries about Android applications.

Table 7: Evaluation of Apposcopy on malware from the Android Malware Genome project.

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

Finally, we remark that Apposcopy's analysis time on these malicious applications is moderate, with an average of 275 seconds per analyzed application containing 18,200 lines of Dalvik bytecode on average.

5.2 Evaluation on Google Play Apps

In a second experiment, we evaluate Apposcopy on thousands of apps from Google Play. Since these applications are available through the official Android market rather than less reliable third-party app markets, we would expect a large majority of these applications to be benign. Hence, by running Apposcopy on Google Play apps, we can assess whether our high-level signatures adequately differentiate benign applications from real malware.

In our experiment, among the 11,215 apps analyzed by Apposcopy, only 16 of them were reported as malware. Specifically, Apposcopy reported two applications to be instances of DroidDreamLight, one to be an instance of DroidDream and another one to be an instance of Pjapps. The remaining 12 applications were categorized as DroidKungFu. To decide whether these 16 apps are indeed malware, we uploaded them to VirusTotal [5] for analyzing suspicious applications. VirusTotal is a service that runs multiple anti-virus tools on the uploaded application and shows their aggregate results. Based on the results provided by VirusTotal, the majority of anti-virus tools agree with Apposcopy's classification for 13 of the 16 reported malware. For the remaining three applications, the majority of the tools classify them as malicious adware while Apposcopy classifies them as instances of DroidKungFu. This experiment confirms our claim that Apposcopy does not generate a lot of false alarms and that our malware signatures can distinguish benign applications from real malware.

Similar to the experiments from Section 5.1, Apposcopy takes an average of 346 seconds to analyze a Google Play application with 26,786 lines of Dalvik bytecode on average.

5.3 Evaluation on Obfuscated Apps

To substantiate our claim that Apposcopy is resilient to code transformations, we compare the detection rate of Apposcopy with other anti-virus tools on obfuscated versions of known malware. For this experiment, we obfuscated existing malware using the ProGuard tool [2], which is commonly used by malware writers to evade detection. In addition, since the databases of some of the anti-virus tools include signatures of malware samples obfuscated by ProGuard, we also applied three additional obfuscations that are not performed by ProGuard: First, our obfuscator changes the names of components, classes, methods, and fields. Second, all invocations to methods of `android.*` classes are redirected through proxy methods. Third, our obfuscator also performs string encryption, including encryption of component names as well as action and data type values of intents.

Table 8 shows the results of our third experiment on obfuscated malware. Each row in this table corresponds to an application that is an instance of a known malware family (and whose unobfuscated version can be identified as malware by all tools considered in our evaluation). Each column in the table corresponds to a leading anti-virus tool, namely, AVG, Symantec, ESET, Dr. Web, Kaspersky, Trend Micro, and McAfee. A check (✓) indicates that the tool is able to detect the obfuscated version of the program as malware, and a cross (✗) means that the tool is unable to classify the obfuscated version as malicious. As Table 8 shows, Apposcopy is resistant to these obfuscations for all malware that we considered. In contrast, none of the other tools can successfully classify all of the obfuscated apps as malware.

5.4 Comparison with Kirin

In addition to comparing Apposcopy with commercial anti-virus tools, we also compared Apposcopy against Kirin [19], which is the *only* publicly available research tool for Android malware detection. As explained in Section 6, Kirin is a signature-based malware detector that classifies an app as malware if it uses a dangerous combination of permissions specified by the malware signature. On the set of malicious apps considered in Section 5.1, Kirin reports only 532 apps out of 1,027 malicious apps to be malware. This corresponds to a false negative rate of 48% , which is quite high compared to the 10% false negative rate of Apposcopy. On the other hand, for the set of applications taken from Google Play and considered in Section 5.2, Kirin reports 8% of these apps to be malware, while Apposcopy classifies only 0.14% of these apps as malicious. We manually inspected 20 out of the 886 apps classified as malware by the Kirin tool and also compared with the results of VirusTotal. Our evaluation revealed that the overwhelming majority of the apps classified as malware by Kirin are false positives. Hence, our experiments demonstrate that Apposcopy outperforms Kirin both in terms of false positives as well as false negatives.

6. RELATED WORK

Taint analysis. Both *dynamic* and *static* taint analyses have been proposed for tracking information-flow in mobile applications. For example, TaintDroid [17] and VetDroid [37] are dynamic taint analyses that track information flow by instrumenting the Dalvik VM, and examples of static taint analyses include [20, 21, 23, 16]. While Apposcopy employs static taint analysis as one of its components, we observe that not every application that leaks sensitive data is malicious — in fact, many benign apps transmit sensitive data for performing their required functionality. Thus, taint analyses on their own are not sufficient for automatically differentiating malicious apps from benign apps, and we propose to combine taint analysis with high-level malware signatures to effectively identify malicious code.

Signature-based malware detection. Many techniques for identifying existing malware are signature-based, meaning that they look for patterns identifying a certain malware family. In its simplest form, these patterns are sequences of bytes or instructions [25]. Since such syntactic patterns can be defeated by semantics-preserving transformations, previous work has considered *semantics-aware* malware detection [13]. Similar to [13], Apposcopy detects malware based on their semantic rather than syntactic characteristics. However, our signatures are much higher-level compared to the templatized instruction sequences used in [13] and allow directly specifying control- and data-flow properties of Android applications. Furthermore, the underlying signature matching techniques are also very different.

A popular signature-based malware detection technique for Android is the Kirin tool [19]. The malware signatures in Kirin specify dangerous combinations of Android permissions, and Kirin decides if an application matches a signature by analyzing its manifest file. As demonstrated in our experimental evaluation, Kirin yields many more false positives and false negatives compared to Apposcopy.

Another related approach is the *behavioral detection* technique described in [9]. In that approach, one specifies common malware behavior using temporal logic formulas. However, a key difference between Apposcopy and behavioral detection is that our techniques are purely static, while [9] requires monitoring behavior of the application at run-time.

The DroidRanger tool [40] uses *permission-based behavioral footprint* to detect instances of known malware families. Behavioral footprints include characteristic malware features, such as listening to certain system events, calling suspicious APIs, and containing hard-coded strings. While these behavioral footprints can be viewed as high-level malware signatures, they differ from those of Apposcopy in several ways: Behavioral footprints neither refer to information flow properties of an application nor do they express control flow dependencies between different components. Furthermore, behavioral footprints include hard-coded string values, which are easy to obfuscate by malware writers.

Zero-day malware detection. A known limitation of signature-based approaches, including Apposcopy, is that they can only detect instances of known malware families. In contrast, zero-day malware detectors try to uncover unknown malware families. For example, RiskRanker [24] performs several risk analyses to rank Android applications as high-, medium-, or low-risk. These risk analyses include techniques to identify suspicious code that exploits platform-level vulnerabilities or sends private data without being triggered by user events. In addition to identifying instances of known malware, DroidRanger [40] also tries to uncover zero-day malware by performing heuristic-based filtering to identify certain “inherently suspicious” behaviors.

Many recent malware detectors, such as [8, 22, 10, 33, 6], use machine learning to detect zero-day malware. For example, Drebin [8] performs light-weight static analysis to extract features, such as permissions and API calls, and trains

Table 8: Comparison between Apposcopy and other tools on obfuscated malware.

Family	AVG	Symantec	ESET	Dr. Web	Kaspersky	Trend Micro	McAfee	Apposcopy
DroidKungFu	✗	✗	✓	✗	✓	✗	✗	✓
Geinimi	✗	✗	✗	✗	✗	✗	✗	✓
DroidDreamLight	✗	✗	✗	✗	✗	✗	✗	✓
GoldDream	✗	✗	✗	✗	✗	✓	✗	✓
DroidDream	✓	✓	✓	✓	✓	✓	✓	✓
BeanBot	✗	✗	✗	✗	✗	✗	✗	✓
GingerMaster	✗	✗	✓	✓	✓	✓	✓	✓
Pjapps	✗	✗	✗	✗	✗	✗	✗	✓
Bgserv	✓	✗	✗	✗	✓	✗	✗	✓
CoinPirate	✗	✗	✗	✗	✓	✗	✗	✓
jSMSHider	✓	✓	✓	✓	✓	✓	✓	✓
AnserverBot	✓	✗	✓	✓	✓	✓	✓	✓
DroidCoupon	✗	✗	✓	✓	✓	✗	✗	✓
ADRD	✗	✗	✗	✗	✗	✗	✗	✓
Success rate	28.6%	14.3%	42.9%	35.7%	57.1%	35.7%	28.6%	100.0%

an SVM to find a hyperplane separating benign apps from malware. The DroidAPIMiner tool [6] also considers API features and uses machine learning to automatically classify an application as malicious or benign. While learning-based approaches are powerful for detecting unknown malware, their precision relies on representative training sets.

We believe all of these zero-day malware detection techniques are complementary to Apposcopy: While Apposcopy can identify instances of known malware families with few false alarms, zero-day malware detectors can help uncover new malware families, albeit at the cost of more false positives or more involvement from a security auditor.

Static analysis for malware detection. Static analysis and model checking have been used to detect security vulnerabilities for a long time. In the context of mobile malware, the SAAF tool [26] uses program slicing to identify suspicious method arguments, such as certain URLs or phone numbers. The work described in [18] performs various static analyses, including taint analysis, to better understand smartphone application security in over 1,000 popular Android apps. One of the interesting conclusions from this study is that, while many apps misuse privacy-sensitive information, few of these apps can be classified as malware.

The Pegasus system [11] focuses on malware that can be identified by the order in which certain permissions and APIs are used. The user writes specifications of expected app behavior using temporal logic formulas, and Pegasus model checks these specifications against an abstraction called the *permission event graph (PEG)*. Pegasus differs from our approach in that (i) it targets a different class of malware, and (ii) PEG abstracts the relationship between the Android system and the application, while ICCG abstracts the relationship between different components in the application.

Some recent papers address the detection of re-packaged apps which often inject adware or malicious features into legitimate apps [39, 38, 15, 14]. While some repackaged apps may contain malware, these techniques mainly focus on clone rather than malware detection.

Analysis of ICC. ComDroid [12] analyzes ICC of Android apps to expose security vulnerabilities, such as intent spoofing or unauthorized intent receipt. CHEX [29] performs static analysis to identify app entry points and uses this information to detect component hijacking vulnerabilities. In contrast to Apposcopy, ComDroid and CHEX are meant to be used by developers to identify security vulnerabilities in their own applications.

Epicc [32] also addresses ICC in Android and proposes a static analysis for inferring *ICC specifications*. These specifications include ICC entry and exit points, information about the action, data and category components of intents used for ICC, as well as Intent key/value types. While our ICCG encodes similar information to the specifications inferred by Epicc, we show that the ICCG is a useful abstraction for specifying and identifying Android malware.

7. LIMITATIONS

Like any signature-based solution, Apposcopy is not invincible; it is very hard to design any signature-based scheme that cannot be defeated by a suitably designed automatic obfuscator. In particular, similar to any static analysis based system, Apposcopy may be defeated by obfuscation techniques such as dynamic code loading and use of reflection in combination with obfuscation of method or class names. However, such attempts to escape detection are likely to be deemed suspicious and may invite further scrutiny.

Second, since Apposcopy performs deep static analysis to uncover semantic properties of an app, it may be unfit for scenarios that require instant detection of malware. However, smartphone apps are generally distributed through centralized app stores, which enables deployment of Apposcopy to scan apps as they are submitted to the app store.

8. CONCLUSION AND FUTURE WORK

We presented Apposcopy, a static analysis approach for detecting malware in the mobile apps ecosystem. Malware that belong to one family share a common set of characteristic behaviors, which an auditor can encode through Apposcopy's Datalog-based malware specification language. Apposcopy performs deep static analysis to extract data-flow and control-flow properties of Android applications and uses these results to identify whether a given application belongs to a known malware family. Our experiments indicate that Apposcopy can detect malware with high accuracy and that its signatures are resilient to various program obfuscations.

There are several opportunities for future work. We will develop techniques to improve the efficiency and precision of Apposcopy's static analyses. We also plan to develop techniques to automatically de-obfuscate apps to enhance Apposcopy's resilience to some types of obfuscations (see Section 7). Finally, we plan to develop techniques to automatically learn malware signatures from a set of apps labeled with their corresponding malware family (or as benign).

9. REFERENCES

[1] Android malware genome project. http://www.malgenomeproject.org/.

[2] ProGuard. http://proguard.sourceforge.net/.

[3] Q2 IT evolution threat report. http://www.securelist.com/en/analysis/204792299/IT_Threat_Evolution_Q2_2013.

[4] US homeland security report. http://info.publicintelligence.net/DHS-FBI-AndroidThreats.pdf.

[5] VirusTotal. https://www.virustotal.com/en/.

[6] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-level features for robust malware detection in Android. In *SecureComm*, 2013.

[7] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[8] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. 2014.

[9] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *MobiSys*, pages 225–238, 2008.

[10] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: triage for market-scale mobile malware analysis. In *WISEC*, pages 13–24, 2013.

[11] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.

[13] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy*, pages 32–46, 2005.

[14] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on Android markets. In *ESORICS*, pages 37–54. 2012.

[15] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar Android applications. In *ESORICS*, 2013.

[16] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[17] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[18] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security Symposium*, 2011.

[19] W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *ACM Conference on Computer and Communications Security*, pages 235–245, 2009.

[20] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for android application. Technical report, EC SPRIDE Technical Report, 2013.

[21] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, November 2009.

[22] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *AISec*, pages 45–54, 2013.

[23] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST*, pages 291–307, 2012.

[24] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *MobiSys*, pages 281–294, 2012.

[25] K. Griffin, S. Schneider, X. Hu, and T. cker Chiueh. Automatic generation of string signatures for malware detection. In *RAID*, pages 101–120, 2009.

[26] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing droids: program slicing for smali code. In *SAC*, pages 1844–1851, 2013.

[27] X. Jiang. Security alert: New Android malware – GoldDream – found in alternative app markets. http://www.csc.ncsu.edu/faculty/jiang/GoldDream/, 2011.

[28] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, pages 423–434, 2013.

[29] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM Conference on Computer and Communications Security*, pages 229–240, 2012.

[30] M. Might, Y. Smaragdakis, and D. V. Horn. Resolving and exploiting the *k*-cfa paradox: illuminating functional vs. object-oriented program analysis. In *PLDI*, pages 305–315, 2010.

[31] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *TOSEM*, 14(1):1–41, 2005.

[32] D. Octeau, P. McDaniel, S. Jha, A. Bartel, , E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium*, 2013.

[33] H. Peng, C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *ACM Conference on Computer and Communications Security*, pages 241–252, 2012.

[34] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: evaluating Android anti-malware against transformation attacks. In *ASIACCS*, pages 329–334. ACM, 2013.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.

[36] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In *APLAS*, pages 97–118, 2005.

[37] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *ACM Conference on Computer and Communications Security*, pages 611–622, 2013.

[38] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *CODASPY*, pages 185–196, 2013.

[39] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *CODASPY*, pages 317–326, 2012.

[40] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.