SCanDroid: Automated Security Certification of Android Applications

University of Maryland Department of Computer Science Technical Report CS-TR-4991, November 2009

Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster

Department of Computer Science, University of Maryland, College Park
E-mail: {afuchs,avik,jfoster}@cs.umd.edu

Abstract

Android is a popular mobile-device platform developed by Google. Android's application model is designed to encourage applications to share their code and data with other applications. While such sharing can be tightly controlled with permissions, in general users cannot determine what applications will do with their data, and thereby cannot decide what permissions such applications should run with. In this paper we present SCANDROID*, a tool for reasoning automatically about the security of Android applications.* SCANDROID*'s analysis is modular to allow incremental checking of applications as they are installed on an Android device. It extracts security specifications from manifests that accompany such applications, and checks whether data flows through those applications are consistent with those specifications. To our knowledge,* SCANDROID *is the first program analysis tool for Android, and we expect it to be useful for automated security certification of Android applications.*

1 Introduction

Android [3] is Google's open-source platform for mobile devices, which is recently enjoying wide adoption by the industry. Designed to be a complete software stack, Android includes an operating system, middleware, and core applications. Furthermore, it comes with an SDK that provides the tools and APIs necessary to develop new applications for the platform in Java [3]. Developers of new applications have full access to the same framework APIs used by the core applications.

Android's application model has several interesting features. First, applications must follow a specific structure, *i.e.*, they must be composed of some basic kinds of components understood by Android. This design encourages sharing of code and data across applications. Next, interactions between components can be tightly controlled. By default, components within an application are sandboxed by Android, and other applications may access such components only if they have the required permissions to do so. This design promises some measure of protection from malicious applications.

However, enforcing permissions is not sufficient to prevent security violations, since permissions may be misused, intentionally or unintentionally, to introduce insecure data flows. Indeed, suppose that Alice downloads and installs a new application, developed by Bob, on her Android-based phone. Say this application, *wikinotes*, interacts with a core application, *notes*, to publish some notes from the phone to a wiki, and to sync edits back from the wiki to the phone. Of course, Alice would not like all her notes to be published, and would not like all her published notes to be edited; for instance, her notes may include details of her ongoing research. How can she know whether it is safe to run the application? Can she trust the application to safely access her data? Conversely, Bob may want to be able to convince Alice that his application can be run safely on her phone.

In this paper, we present SCANDROID,[1] a tool for automated security certification of Android applications. SCANDROID statically analyzes data flows through Android applications, and can make security-relevant decisions automatically, based on such flows. In particular, it can decide whether it is safe for an application to run with certain permissions, based on the permissions enforced by other applications. Alternatively, it can provide enough context to the user to make informed security-relevant decisions. SCANDROID can also be useful in various proof-carrying code (PCC) [11] settings. For example, applications can be reviewed offline with SCANDROID by an application store [2], and Android devices can check certificates of security issued by the application store at install time. Alternatively, the developer can construct a safety proof for the application by using our analysis, and the device can verify that proof before installing the application.

At the heart of SCANDROID is a modular data flow anal-

[1] The name is intended to abbreviate "Security Certifier for anDroid", although various puns might be intended as well.

ysis for Android applications, designed to allow incremental checking of applications as they are installed on an Android device. Our analysis tracks data flows through and across components, while relying on an underlying abstract semantics for Android applications. The data flows can be fairly complicated, due to sophisticated protocols run by Android to route control between components. Our abstract semantics for Android applications exposes these control routes to our analysis.

We formalize the basic elements of our data flow analysis as a constraint system, based on an existing core calculus to describe and reason about Android applications [7]. We show how end-to-end security can be enforced with our data flow analysis. In our formalism, we focus only on constructs that are unique to Android, while ignoring the other usual Java constructs that may appear in Android applications. This simplification allows us to study Android-specific features in isolation. Our system relies on the access control mechanisms already provided by Android, and enforces "best practices" for developing secure applications with these mechanisms. The resulting guarantees include standard data-flow security properties for well-constrained applications described in the calculus.

Next, we extend and implement this core analysis to reason about actual Android applications. For this purpose, we must consider the usual Java constructs in combination with Android-specific constructs. This poses some significant challenges: for instance, we need a string analysis to recover addresses of components; we need a pointer analysis to track flows through the heap; we need to handle interprocedural flows through JVML bytecode, and so on. Our implementation is built over WALA, an open-source collection of libraries for Java program analysis. Although we have not proved so, we expect that our formal guarantees should carry over to our implementation.

In sum, our contributions in this paper are as follows.

- We design a modular data flow analysis for Android applications, which tracks data flows through and across components, while relying on an abstract semantics for Android applications.

- We formalize the basic elements of our analysis in a core calculus to describe and reason about Android applications. We prove that applications verified by our analysis have standard data-flow security properties.

- We present a tool, SCANDROID, that extends and implements this analysis to reason about actual Android applications. Although we have not proved so, we expect that our formal guarantees can be carried over to our implementation.

- We indicate how SCANDROID can be useful for automated security certification of Android applications.

To our knowledge, SCANDROID is the first program analysis tool for Android. While we focus on security in this paper, we believe that SCANDROID's data flow analysis will be useful in other contexts as well. For instance, it may be used to prove the safety of type casts, which are common for Android's data structures; it may also be used to infer minimal sets of permissions required for functionality. At the very least, we believe that this setting provides an ideal opportunity to bring language-based security to the mainstream: the idea of certified installation should certainly be attractive to a growing and diverse Android community.

The remainder of the paper is organized as follows. In Section 2 we present an overview of Android, focusing on its application model, data structures, and security mechanisms. In Section 3, we provide a series of examples to illustrate data flows through Android applications. In Section 4, we formalize our analysis and outline its security guarantees. In Section 5, we describe the details of our implementation. In Section 6, we discuss limitations and future extensions. In Section 7, we discuss some closely related work. Finally, in Section 8, we conclude.

2 Overview of Android

Android implements a complete software stack for running mobile device applications. At the lowest layer is a Linux kernel that provides device drivers, memory management, power management, and networking. Next, Android provides some native libraries for graphics, database management, and web browser functionality, which can be called through Java interfaces. Furthermore, Android includes core Java libraries, and a virtual machine for running customized bytecode (called *dex*) derived from JVML bytecode. Above this layer is the application framework, which serves as an abstract machine for applications. Finally, the top layer contains application code, developed in Java with the APIs provided by an SDK.

Next, we describe Android's application model, focusing on how applications are developed and run in detail.

2.1 Application components

An Android application is a package of *components*, each of which can be instantiated and run as necessary (possibly even by other applications) [3]. Components are of the following types:

Activity components form the basis of the user interface. Usually, each window of the application is controlled by some activity.

Service components run in the background, and remain active even if windows are switched. Services can expose interfaces for communication with other applications.

BroadcastReceiver components react asynchronously to messages from other applications.

ContentProvider components store data relevant to the application, usually in a database. Such data can be shared across applications.

Consider, for example, a music-player application for an Android-based phone. This application may include several components. There may be activities for viewing the songs on the phone and for editing the details of a particular song. There may be a service for playing a song in the background. There may be broadcast receivers for pausing a song when a call comes in, and for restarting the song when the call ends. Finally, there may be a content provider for sharing the songs on the phone.

The Android SDK provides a base class for each type of component, with callback methods that are run at various points in the life cycle of the associated component. Each component of an application is defined by extending one of the base classes and overriding the methods in that class. (The methods have default implementations, so it is not necessary to override all such methods in the class.) The following is a simplified version of the interfaces defined for each of the four component types. (The types Bundle, Intent, Cursor, ContentValues, Uri, and IBinder describe data structures that are used to pass data through and between components; we will explain these data structures later.)

```
1  class Activity ... {
2    public void onCreate(Bundle ...) {...}
3    public void onActivityResult(..., Intent data) {...}
4    ...
5  }
6  class Service ... {
7    public IBinder onBind(Intent data) {...}
8    ...
9  }
10 class BroadcastReceiver ... {
11   public void onReceive(..., Intent data) {...}
12   ...
13 }
14 class ContentProvider ... {
15   public Cursor query(Uri queryUri, ...) {...}
16   public int update(Uri updateUri, ContentValues data, ...) {...}
17   ...
18 }
```

Effectively, each of these methods is an entry point (*i.e.*, a main method) that can be called at various points in the lifecycle of the component. Our implementation carries out a modular analysis over all such entry points. However, in this paper we focus only on the methods shown above for simplicity.

2.2 Environment

Each Android application publishes its contents to the framework via a *manifest*. The manifest contains a list of components that are hosted by an application, some information used to link components at run time, and information about the permissions enforced and requested by the application. The manifests for all the applications installed on a device together form what we will call the environment. The environment affects Android's run-time decisions about control flow and data flow between application components, and is therefore a critical consideration for any flow analysis.

2.3 Data structures

We now explain some of the data structures that appear in the signatures above. In Android, data is usually passed around as hashes of key/value pairs. Containers for such data include Bundles, Cursors, and ContentValues (and there are others, *e.g.*, to share preferences across components in an application). Sometimes, it is convenient to tag data with an address, especially to construct messages for inter-component communication. Containers for such messages, *i.e.*, data/address pairs, include Intents. Addresses for data can be specified as Uris. Finally, IBinder is a higher-order data structure that may contain Java interfaces for inter-process communication on Android.

2.4 Inter-component communication

Besides managing the lifecycles of individual components, Android's application framework is also responsible for calling components from other components and for passing messages between components. In application code, these interactions occur through various method calls and callback methods. In particular, *inflow* methods are used to import data to the application, and *outflow* methods are used to export data from the application. Some of the inflow methods are shown in the interfaces above, *e.g.*, ContentProvider.query and Activity.onActivityResult; there are others that are not shown, such as Activity.getIntent. In general, each inflow method can be matched with an outflow method. For instance, Activity.getIntent() is matched by Activity.startActivityForResult(), Activity.onActivityResult() is matched by Activity.setResult(), and ContentProvider.query is matched by ContentProvider.update().

Much of the complexity of orchestrating inter-component interactions lies in linking outflow methods with components that provide the corresponding inflow methods. As explained above, addresses can be specified either as Uris to address ContentProviders or inside Intents to address Activitys, Services, or BroadcastReceivers.

Furthermore, additional linking information is provided in the manifest.

Ultimately, addresses influence how control flows across components, and permissions (described later) can restrict such flows based on those addresses. Thus, our analysis needs to track Uris and Intents very precisely. Next we discuss the addressing mechanisms through Uris and Intents in more detail.

Uri-based addressing In application code, query and update calls to ContentProviders must be addressed through Uris. A ContentProvider is specified in the manifest along with the address of its *authority*. For example, a ContentProvider that shares code certificates from some domain might have domain.certificates as its authority. A query or update on an Uri of the form content://domain.certificates/... would then be directed to that ContentProvider.

Intent-based addressing On the other hand, components such as Activitys, Services, or BroadcastReceivers must be addressed through Intents. For example, Intents are passed as parameters in inter-component calls for launching activities (via startActivityForResult), binding to services (via bindService), and sending messages to broadcast receivers (via sendBroadcast). An Intent can specify the component to call either by name (*e.g.*, "domain.services.PostBlog"), or more abstractly by action (*e.g.*, "POSTBLOG"). As explained above, Intents may also carry data, usually in a Bundle.

If the address in an Intent is an action rather than a name, the Intent is matched against *intent filters* that are specified alongside component descriptions in the manifest. The action specified in the Intent is checked against the action specified in the intent filter to find a suitable component for processing the call.

Finally, we should point out that some components, such as BroadcastReceivers and Services, may be constructed dynamically by application code. As such, linking information about such components is provided at run time in their constructors, rather than statically in the manifest. More generally, components that handle events in Android, such as notification managers and user interfaces, follow a similar pattern. In our formal analysis, we conveniently generalize such components as *listeners*.

2.5 Security mechanisms

An application can share its data and functionality with other applications by letting such applications access its components. Clearly, these accesses must be carefully controlled for security. Android provides the following key access control mechanisms [3].

Permissions Any application needs explicit permissions to access the components of other applications. Crucially, *such permissions are set at install time, not at run time*. An application's manifest must declare any permissions that the application may require. The package installer sets these permissions via dialog with the user. No further decisions are made at run time; if the application's code needs a permission at run time that is not set at install time, it blocks, and its resources are reclaimed by Android.

Conversely, an application' manifest declares any permissions required by other applications to access its components. For dynamic components, this information is provided in their constructors. In particular, an application can enforce permissions for launching its activities, binding to its services, sending messages to its broadcast receivers or receiving messages from any of its components, and querying and updating data stored by its content providers.

Isolation and Signatures Going further, Android derives several protection mechanisms from the Linux kernel. Every application runs in its own Linux process, on its own VM. Android starts the application's process when any of the application's code needs to be run, and stops the process when another application's code needs to be run. The application's code runs in isolation from the code of all other applications. Finally, each application is assigned a unique Linux UID, so the application's files are not visible to other applications.

That said, it is possible for several applications to share, through signatures, the same UID (see below), in which case their files become visible to each other. Such applications can also run in the same process, sharing the same VM. In particular, any Android application must be signed with a certificate whose private key is held by the developer. The certificate does not need to be signed by a certificate authority; it is used only to establish trust between applications by the same developer. Thus, such applications may share the same UID or the same permissions.

3 Examples

In this section we show a series of examples to illustrate how data may flow through Android applications. In particular we are concerned about flows between Uris, which address content providers, since ultimately they are the data sources and sinks across applications.

Intra-component flows We begin with an example of a simple flow from one Uri to another within an activity.

Example (i)

```
19 public class Util {
20   private static String authorityA = "some.authority";
```

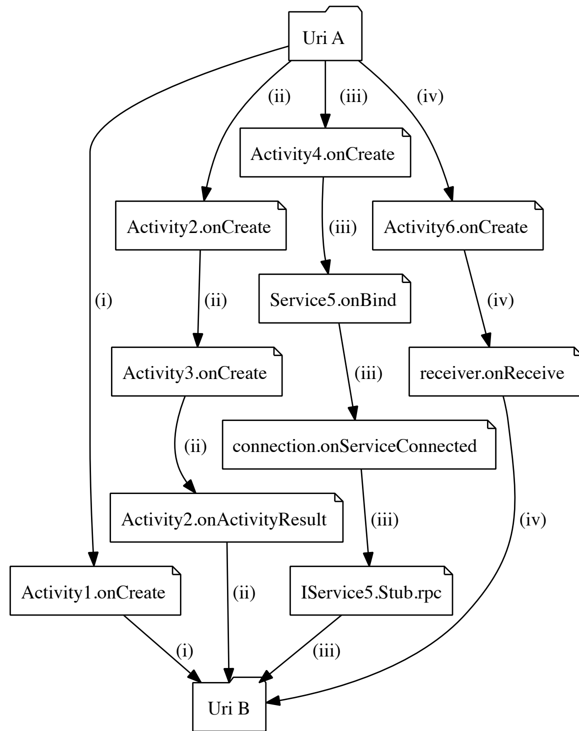

Figure 1. Data flows from Uri A to Uri B

```
21     private static String authorityB = "some.other.authority";
22
23     public static final Uri uriA = Uri.parse("content://"+authorityA);
24     public static final Uri uriB = Uri.parse("content://"+authorityB);
25
26     public static String readA(ContentResolver handle) {
27       Uri queryUri = Uri.withAppendedPath(uriA, "param");
28       Cursor result = handle.query(queryUri, ...);
29       return result.toString();
30     }
31
32     public static void writeB(ContentResolver handle, String s) {
33       Uri updateUri = uriB;
34       ContentValues values = new ContentValues();
35       values.put("param", data);
36       handle.update(updateUri, values, ...);
37     }
38   }
39
40   public class Activity1 extends Activity {
41     public void onCreate(...) {
42       ...
43       String s = Util.readA(getContentResolver());
44       Util.writeB(getContentResolver(), s);
45     }
46   }
```

The class Util defines some methods that will be used throughout this section. One of these methods, readA (lines 27–31), is used to issue a query on the Uri content://some.authority/param (line 29), which we will call Uri A; this returns a Cursor object over some data at that Uri (line 30). The other method, writeB (lines 33–38), is used to package such data into a ContentValues object (line 36) and send it off to update the contents at the Uri content://some.other.authority (line 37), which we will call Uri B. (Both of these methods take a ContentResolver object as parameter, which can be obtained simply by calling a built-in method getContentResolver.)

Suppose that we launch an instance of Activity1. This passes control to the onCreate method of that activity (line 42). Next, readA is called to query Uri A (line 44), and writeB is called with the result to update Uri B (line 45). Thus, we have a flow of data from Uri A to Uri B; this flow is shown via the edges marked (i) in the graph in Figure 1.

Such a flow may have unintended security consequences.

- Suppose that the data at Uri A is intended to be secret, *i.e.*, reading data at Uri A requires permission $\top$. At the same time, suppose that the data at Uri B is public, *i.e.*, reading data at Uri B requires permission $\bot$. Clearly this admits a secrecy violation, because anybody who has $\bot$ permission can effectively read data from Uri A.

- Dually, suppose that the data at Uri B is intended to be trusted, *i.e.*, writing data at Uri B requires permission $\top$. At the same time, suppose that the data at Uri A is tainted, *i.e.*, writing data at Uri A requires permission $\bot$. Clearly this admits an integrity violation, because anybody who has $\bot$ permission can effectively write data to Uri B.

To prevent such violations, our analysis enforces that the permission required to read Uri B should be at least as high as the permission required to read Uri A; and conversely, the permission required to write Uri A should be at least as high as the permission required to write Uri B.

Inter-component flows Our next example involves a similar overall flow from Uri A to Uri B, but split across multiple activities, *i.e.*, the overall flow is a transitive composition of simpler flows within and through these activities.

Example (ii)

```
47   public class Util {
48     ...
49     public static Intent pack(String s) {
50       Intent intent = new Intent();
51       intent.putExtra("param", s);
52       return intent;
53     }
54
55     public static String unpack(Intent data) {
56       return data.getStringExtra("param");
57     }
58   }
```

```
59 public class Activity2 extends Activity {
60   public void onCreate(...) {
61     ...
62     String s = Util.readA(getContentResolver());
63     Intent intent = Util.pack(s);
64     intent.setClass(this, Activity3.class);
65     startActivityForResult(intent, ...);
66   }
67
68   public void onActivityResult(..., Intent data) {
69     String s = Util.unpack(data);
70     Util.writeB(getContentResolver(), s);
71   }
72 }
73
74 public class Activity3 extends Activity {
75   public void onCreate(...) {
76     ...
77     Bundle input = getIntent().getExtras();
78     Intent data = new Intent();
79     i.putExtras(input);
80     setResult(..., i);
81   }
82 }
```

The class Util defines the following new methods: pack, which bundles data in an intent (line 52) and returns it, and unpack, which extracts data from such an intent (line 57) and returns it.

Suppose that we launch an instance of Activity2. As usual, control passes to the onCreate method (line 61), where readA is called to issue a query on Uri A (line 63). Next, pack is called to bundle the returned data into an Intent object (line 64), and an instance of Activity3 is launched with it (lines 65–66). This passes control to the onCreate method of that activity (line 76), where the data in the intent is extracted and bundled back into another intent (lines 78–80), and that intent is set as the result of this activity (line 81). Control now passes back to the onActivityResult method of the calling activity (line 69), where unpack is called to extract the data in the returned intent (line 70), and finally writeB is called to, as usual, package the data and use it to update the contents at Uri B (line 71).

Thus, effectively, we have another flow of data from Uri A to Uri B; this flow is shown via the edges marked (ii) in the graph on the left in Figure 1. This flow may have the same unintended security consequences as in the previous example, so our analysis enforces the same constraints on the permissions required for reading from and writing to Uri A and Uri B, to account for this flow.

Flows through dynamic components Of course, transitive flows may not only involve activities, but also other components. Tracking data flow across components is already nontrivial since control flow across such components is orchestrated implicitly, via reflection, by Android's application framework. Further complications arise when the components involved in such flows are dynamic, rather than static. Our next example involves data flow through a service that dynamically exposes an RPC interface.

Example (iii)

```
83  public class Activity4 extends Activity {
84    public void onCreate(...) {
85        ...
86      String s = Util.readA(getContentResolver());
87      Intent intent = Util.pack(s);
88      intent.setClass(this, Service5.class);
89      ServiceConnection connection = new ServiceConnection() {
90        public void onServiceConnected(..., IBinder service) {
91          IService5.Stub.asInterface(service).rpc();
92        }
93        ...
94      }
95      bindService(intent, connection, ...);
96    }
97  }
98
99  public interface IService5 extends IInterface {
100   // autogenerated from IService5.aidl
101   ...
102   public void rpc();
103 }
104
105 public class Service5 extends Service {
106   public IBinder onBind(final Intent data) {
107     return new IService5.Stub() {
108       public void rpc() {
109         String s = Util.unpack(data);
110         Util.writeB(getContentResolver(), s);
111       }
112     }
113   }
114 }
```

Let us fast-forward to line 88, where data has been read from Uri A and bundled into an intent. This intent is used to bind to an instance of Service5 (line 89—96). Binding to a service actually involves a complicated protocol than simply connecting to the service; in particular, it allows arbitrary RPC calls into the service. This requires passing a dynamic ServiceConnection object (lines 90–95) together with the intent at binding time. Control then passes to the onBind method of the service (line 125), which must return an implementation of some RPC interface with which the connection can interact. Such an interface is typically autogenerated from an AIDL (Android Interface Description Language) specification, as shown in lines 100–104; in this example, the interface just exposes a method named rpc. The implementation of this method (lines 109–112) causes the data bundled in the intent of onBind to flow to Uri B. This implementation is implicitly passed to the onServiceConnected method (line 91) of the connection received at binding time, where it is called (line 92) to finally

trigger the flow. The overall flow is shown via the edges marked (iii) in the graph on the left in Figure 1.

Such complicated flows may also involve other dynamic components, including notification managers and user interfaces. In the following example, a dynamically registered broadcast receiver is involved in a similar flow.

Example (iv)

```
115 public class Activity6 extends Activity {
116   public void onCreate(...) {
117     ...
118     BroadcastReceiver listener = new BroadcastReceiver() {
119       public void onReceive(..., Intent intent) {
120         String s = Util.unpack(data);
121         Util.writeB(getContentResolver(), s);
122       }
123     };
124     registerReceiver(listener, new IntentFilter("ACTION"));
125     String s = Util.readA(getContentResolver());
126     Intent intent = Util.pack(s);
127     intent.setAction("ACTION");
128     sendBroadcast(intent);
129   }
130 }
```

A BroadcastReceiver object (lines 119–124) is registered to listen to intents directed at "ACTION" (line 125). Then, data is read from Uri A, bundled into an intent, and broadcast to any receiver listening to "ACTION". In particular, this passes control to the onReceive method of the registered receiver (line 120), whence the data flows to Uri B. The overall flow is shown via the edges marked (iv) in the graph on the left in Figure 1.

These examples illustrate some possible ways for data to flow through application code. The goal of our analysis is to track all such flows between Uris, so as to generate suitable constraints on the permissions enforced by those Uris. In particular, we want to guarantee that those permissions soundly enforce end-to-end security.

4 Formalization

In this section, we formalize our flow analysis in a core calculus that was developed specifically to describe and reason about Android applications. For simplicity, the calculus does not model general classes and methods; instead, it treats component classes and methods as idealized, primitive constructs. Furthermore, it considers permissions as the only mechanisms to control cross-component interactions.

4.1 Syntax and informal semantics

Let there be a partial order $\geq$ over permissions (which are written in uppercase typewriter font, as in PERM). Components are identified by addresses n, which model class names or actions. Values include variables x and a constant any (which stands for any concrete data); we do not consider addresses as values for simplicity. These values suffice for our purposes, since we consider only (explicit) data flows in our analysis—by definition, such flows do not depend on the values of the data involved. The syntax of programs is as follows:

Program syntax

$v ::= x \mid$ any	value
$t ::=$	code
begin(n, v)	launch activity
end(v)	finish activity
listen$(\texttt{SEND}, n, \lambda x.t)$	register listener
send$(\texttt{LISTEN}, n, v)$	send to listener
query(n)	read from provider
update(n, v)	write to provider
let $x = t$ in t'	evaluate
$\Lsh\, t$	fork
$t + t'$	choice
v	result

The meanings of programs are described informally below, after introducing some additional concepts; a formal operational semantics appears elsewhere [7].

A program runs in an *environment* that maps addresses to components. The calculus considers three kinds of components: activities, listeners, and stores. (As argued previously, listeners generalize services, broadcast receivers, notification managers, and various user interfaces; and stores generalize content providers, databases, files, URIs, and other data containers [7].) An environment is derived from the set of applications installed on the system. The syntax of environments is as follows:

Environment syntax

$\kappa ::=$	component
activity$(\texttt{LAUNCH}, \texttt{PERM}, \lambda x.t, \lambda x.t')$	activity
listener$(\texttt{SEND}, \texttt{PERM}, \lambda x.t)$	listener
store$(\texttt{READ}, \texttt{WRITE}, v)$	store
$\mathcal{E} ::= \varnothing \mid \mathcal{E}, n \mapsto \kappa$	environment

- The component activity$(\texttt{LAUNCH}, \texttt{PERM}, \lambda x.t, \lambda x.t')$ describes an activity that enforces permission LAUNCH on its callers, that runs with permission PERM, and whose onCreate and onActivityResult methods are modeled by the functions $\lambda x.t$ and $\lambda x.t'$; the parameter of $\lambda x.t$ is the value used to launch this activity, and the parameter of $\lambda x.t'$ is the value returned to this activity by any other activity launched by it.

- The component $\mathsf{listener}(\mathtt{SEND}, \mathtt{PERM}, \lambda x.t)$ describes a listener (*e.g.*, a service or a broadcast receiver) that enforces permission SEND on its callers, that runs with permission PERM, and whose listener method (*e.g.*, onBind or onReceive) is modeled by the function $\lambda x.t$; the parameter of $\lambda x.t$ is the value sent to this listener.

- The component $\mathsf{store}(\mathtt{READ}, \mathtt{WRITE}, v)$ describes a store (such as a content provider) that enforces permissions READ and WRITE for accessing its contents, and whose (current) content is v.

Furthermore, a program runs with a permission, which is the permission of the context, *i.e.*, the permission of the application that the code belongs to. The program itself may be run on a stack of windows (produced by launching activities) or in a pool of threads (produced by forking).

- The program $\mathsf{begin}(n, v)$ checks that n is mapped to a component $\mathsf{activity}(\mathtt{LAUNCH}, \mathtt{PERM}, \lambda x.t, \lambda x.t')$, and that the current context has permission LAUNCH. A new window is pushed on the stack, and the program t is run with permission PERM and with x bound to v.

- Conversely, $\mathsf{end}(v)$ pops the current window off the stack and returns control to the previous activity; if that activity is $\mathsf{activity}(\mathtt{LAUNCH}, \mathtt{PERM}, \lambda x.t, \lambda x.t')$, the program t' is run with permission PERM and with x bound to v.

- The program $\mathsf{listen}(\mathtt{SEND}, n, \lambda x.t)$ maps n to the component $\mathsf{listener}(\mathtt{SEND}, \mathtt{PERM}, \lambda x.t)$ in the environment; here, PERM is the permission of the current context.

- Conversely, $\mathsf{send}(\mathtt{LISTEN}, n, v)$ checks that n is mapped to a component $\mathsf{listener}(\mathtt{SEND}, \mathtt{PERM}, \lambda x.t)$, that the current context has permission SEND, and that PERM includes LISTEN. The program t is run with permission PERM, and with x bound to v.

- The program $\mathsf{query}(n)$ checks that n is mapped to a component $\mathsf{store}(\mathtt{READ}, \mathtt{WRITE}, v)$, and that the current context has permission READ; the value v is returned.

- Conversely, $\mathsf{update}(n, v)$ checks that n is mapped to a component $\mathsf{store}(\mathtt{READ}, \mathtt{WRITE}, v')$, and that the current context has permission WRITE; n is then mapped to $\mathsf{store}(\mathtt{READ}, \mathtt{WRITE}, v)$ in the environment.

- The program $\mathsf{let}\ x = t\ \mathsf{in}\ t'$ evaluates t, and then evaluates t' with x bound to the result; x is a local variable.

- The program $\Lsh\ t$ forks a thread that runs t. Forking allows connection to services in the background.

- The program $t + t'$ evaluates either t or t'. Choice allows nondeterministic interaction with user interfaces.

4.2 Some encoded examples

Below we encode some of the examples of Section 3 in this calculus. Our intention is to illustrate that these encodings can serve as convenient intermediate representations for analysis of actual bytecode. In particular, the concise syntax makes the flows in these examples easy to identify.

Recall example (ii), which involved an inter-component flow across two activities. For simplicity, we assume that both activities run with and enforce $\top$ permissions.

$$\begin{aligned}
\mathcal{E}(\mathsf{Activity2}) = \mathsf{activity}(&\top, \top,\\
&\lambda_.\ \mathsf{let}\ x = \mathsf{query}(\mathsf{uriA})\ \mathsf{in}\ \mathsf{begin}(\mathsf{Activity3}, x),\\
&\lambda y.\ \mathsf{update}(\mathsf{uriB}, y))
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}(\mathsf{Activity3}) = \mathsf{activity}(&\top, \top,\\
&\lambda z.\ \mathsf{end}(z),\\
&\lambda_.\ _)
\end{aligned}$$

Next, recall example (iv), which involved a similar flow through a dynamic receiver. Once again, for simplicity we assume that the activity as well as the receiver involved in this flow run with and enforce $\top$ permissions.

$$\begin{aligned}
\mathcal{E}(\mathsf{Activity6}) = \mathsf{activity}(&\top, \top,\\
&\lambda_.\ \mathsf{listen}(\top, \mathsf{ACTION},\\
&\qquad \lambda w.\ \mathsf{update}(\mathsf{uriB}, w));\\
&\quad \mathsf{let}\ x = \mathsf{query}(\mathsf{uriA})\ \mathsf{in}\ \mathsf{send}(\top, \mathsf{ACTION}, x))\\
&\lambda_.\ _)
\end{aligned}$$

4.3 Constraint system

We analyze programs by tracking data flows and then checking whether these flows are secure. Recall that we are ultimately interested in flows from stores to stores. However, since flows may be transitively composed, we need to track for each value in a program the set of stores from which it may flow. Thus, our analysis is best described as a constraint system that derives judgments of the form $\Delta \vDash t \mapsto F$, where Δ is a set of hypotheses, t is a program, and F identifies a set of stores from which data may flow to the result of t. If such a judgment can be derived, we say that F collects the sources of t under Δ.

Our constraint system is modular, in the following sense. For any function $\lambda x.t$, we *assume* some set of stores F from which x may flow in order to track flows through t. Conversely, for any call to such a function with value v, we *guarantee* that the set of stores from which v flows is a subset of F. In the calculus, such functions appear in definitions of components such as activities and listeners, so we require appropriate hypotheses for these definitions. Furthermore, we need to approximate the activity stack, to

track flows of values returned from activities to activities that may have launched them.

In sum, hypotheses in our flow judgments may be of the following forms:

- $n \mapsto F$, where n identifies a store and F identifies a set of stores from which contents may flow into n;
- $n \mapsto \langle \mathsf{Send}\colon F\rangle$, where n identifies a listener and F collects the sources of values that may be sent to this listener (via send);
- $n \mapsto \langle {\downarrow}\colon N,\ \mathsf{Begin}\colon F,\ \mathsf{End}\colon F'\rangle$, where n identifies an activity, N identifies a set of activities that may launch this activity, F collects the sources of values that may be used to launch this activity (via begin), and F' collects the sources of values that may be returned to this activity from other activities launched by it (via end);
- $x \mapsto F$, where F identifies a set of stores from which contents may flow into local variable x;
- $\mathsf{self} \mapsto N$, where N identifies a set of activities that may be running the current code.

Given a set of hypotheses Δ, let $\Delta(\alpha)$ retrieve the entry mapped to identifier α in Δ (where α is one of x, n, or self); and let $\Delta(\alpha).\phi$ look up the field ϕ of that entry, if possible (where ϕ is one of Send, $\downarrow$, Begin, or End).

The rules for deriving constraint judgments for programs are shown in Figure 2. By **(Con query)**, the set of stores from which data may flow to a query on n is $\{n\}$. On the other hand, by **(Con update)**, the set of stores from which values v may flow into n via an update must be contained in $\Delta(n)$. Recall that for each store n, $\Delta(n)$ should identify the set of stores from which contents may flow into n; this will ultimately be the basis of our security analysis.

Moving on, by **(Con listen)**, the body of a listener function (such as onReceive) must be well-constrained assuming that the values sent to it flow from $\Delta(n)$.Send. Conversely, by **(Con send)**, values v that may be sent to a listener n must flow from stores in $\Delta(n)$.Send. **(Con let)**, **(Con fork)**, **(Con choice)**, **(Con var)**, and **(Con any)** are straightforward. Finally, by **(Con begin)**, values v that may be used to launch an activity n must flow from stores in $\Delta(n)$.Begin; furthermore, the current activity (collected in N), must be contained in the $\Delta(n).\downarrow$. Conversely, by **(Con end)**, values v that may be returned by the current activity (collected in N) must flow from $\Delta(m)$.End for every m that may have launched the current activity.

Based on constraint judgments for programs, we derive constraint judgments for environments, which are of the form $\Delta \vDash \mathcal{E}$. Such a judgment is defined pointwise, *i.e.*, we have $\Delta \vDash \mathcal{E}$ if and only if for each mapping $n \mapsto \kappa$ in $\mathcal{E}$, we have $\Delta \vDash n \mapsto \kappa$. The rules for deriving the latter judgments are also shown in Figure 2. By **(Con activity)**,

Well-constrained programs $\Delta \vDash t \mapsto F$

$$\textbf{(Con query)}\quad \Delta \vDash \mathsf{query}(n) \mapsto \{n\}$$

$$\textbf{(Con update)}\quad \frac{\Delta \vDash v \mapsto F \qquad F \subseteq \Delta(n)}{\Delta \vDash \mathsf{update}(n, v) \mapsto \{\}}$$

$$\textbf{(Con listen)}\quad \frac{\Delta, x \mapsto \Delta(n).\mathsf{Send} \vDash t \mapsto \{\}}{\Delta \vDash \mathsf{listen}(\mathtt{SEND}, n, \lambda x.t) \mapsto \{\}}$$

$$\textbf{(Con send)}\quad \frac{\Delta \vDash v \mapsto F \qquad F \subseteq \Delta(n).\mathsf{Send}}{\Delta \vDash \mathsf{send}(\mathtt{LISTEN}, (n, v)) \mapsto \{\}}$$

$$\textbf{(Con let)}\quad \frac{\Delta \vDash t \mapsto F \qquad \Delta, x \mapsto F \vDash t' \mapsto F'}{\Delta \vDash \mathsf{let}\ x = t\ \mathsf{in}\ t' \mapsto F'}$$

$$\textbf{(Con fork)}\quad \frac{\Delta \vDash t \mapsto \{\}}{\Delta \vDash\ \Lsh t \mapsto \{\}}$$

$$\textbf{(Con choice)}\quad \frac{\Delta \vDash t \mapsto F \qquad \Delta \vDash t' \mapsto F}{\Delta \vDash t + t' \mapsto F}$$

$$\textbf{(Con var)}\quad \Delta \vDash x \mapsto \Delta(x)$$

$$\textbf{(Con any)}\quad \Delta \vDash \mathsf{any} \mapsto \{\}$$

$$\textbf{(Con begin)}\quad \frac{\begin{array}{cc}\Delta \vDash v \mapsto F & \Delta(\mathsf{self}) = N \\ F \subseteq \Delta(n).\mathsf{Begin} & N \subseteq \Delta(n).\downarrow\end{array}}{\Delta \vDash \mathsf{begin}(n, v) \mapsto \{\}}$$

$$\textbf{(Con end)}\quad \frac{\begin{array}{c}\Delta \vDash v \mapsto F \qquad \Delta(\mathsf{self}) = N \\ \forall n \in N.\, \forall m \in \Delta(n).\downarrow\colon\ F \subseteq \Delta(m).\mathsf{End}\end{array}}{\Delta \vDash \mathsf{end}(v) \mapsto \{\}}$$

Well-constrained environment $\Delta \vDash \mathcal{E}$, defined pointwise

$$\textbf{(Con activity)}\quad \frac{\begin{array}{c}\Delta, \mathsf{self} \mapsto \{n\}, x \mapsto \Delta(n).\mathsf{Begin} \vDash t \mapsto \{\} \\ \Delta, \mathsf{self} \mapsto \{n\}, x \mapsto \Delta(n).\mathsf{End} \vDash t' \mapsto \{\}\end{array}}{\Delta \vDash n \mapsto \mathsf{activity}(\mathtt{LAUNCH}, \mathtt{PERM}, \lambda x.t, \lambda x.t')}$$

$$\textbf{(Con listener)}\quad \frac{\Delta, \mathsf{self} \mapsto \mathbb{N}, x \mapsto \Delta(n).\mathsf{Send} \vDash t \mapsto \{\}}{\Delta \vDash n \mapsto \mathsf{listener}(\mathtt{SEND}, \mathtt{PERM}, \lambda x.t)}$$

$$\textbf{(Con provider)}\quad \frac{\Delta \vDash v \mapsto F \qquad F \subseteq \Delta(n)}{\Delta \vDash n \mapsto \mathsf{store}(\mathtt{READ}, \mathtt{WRITE}, v)}$$

Figure 2. Constraint system

the bodies of the onCreate and onActivityResult functions of an activity n must be well-constrained assuming that the set of activities that may run these functions is $\{n\}$, and the values passed to these functions flow from $\Delta(n)$.Begin and $\Delta(n)$.End. **(Con listener)** is similar to **(Con listen)**, except that the set of activities that may run the listener function is unknown, and is conservatively taken to be the set of all activities $\mathbb{N}$. In particular, this means that for any activity m launched by a listener, $\Delta(m).\downarrow$ must be unknown, and our analysis must consider values returned by m to flow to the onActivityResult methods of all activities. While in general this may introduce imprecision, in our experience it typically does not since activities launched by listeners (such as user interfaces) seldom return interesting values, *i.e.*, the flow sets for such values is often $\{\}$.

In our implementation (described in the next section), we essentially derive these flow judgments by inference, by initializing each of flow sets in the hypotheses to $\varnothing$ and then computing a fixpoint of these sets by applying the derivation rules in reverse. The solution of the constraint system is then the least set of consistent hypotheses Δ.

Finally, we analyze these hypotheses in conjunction with the environment for security. (Recall that an environment essentially describes the state of an Android-based device.) Formally, we consider environment $\mathcal{E}$ to be *data flow secure* if, for some set of hypotheses Δ, we have $\Delta \vDash \mathcal{E}$, such that whenever $m \in \Delta(n)$, $\mathcal{E}(m) = \mathsf{store}(\mathtt{READ}, \mathtt{WRITE}, _)$, and $\mathcal{E}(n) = \mathsf{store}(\mathtt{READ}', \mathtt{WRITE}', _)$, we have $\mathtt{READ}' \geq \mathtt{READ}$ and $\mathtt{WRITE} \geq \mathtt{WRITE}'$. In other words, $\mathcal{E}$ is data flow secure if the permissions enforced in $\mathcal{E}$ are consistent with the flows through $\mathcal{E}$, as inferred by solving $\Delta \vDash \mathcal{E}$ for Δ.

4.4 The encoded examples, revisited

Let us run our analysis on the examples encoded above. Example (ii) is well-constrained under the following hypotheses.

$\Delta(\mathsf{uriA}) = \{\}$
$\Delta(\mathsf{Activity2}) = \langle \downarrow: \{\}, \mathsf{Begin}: \{\}, \mathsf{End}: \{\mathsf{uriA}\}\rangle$
$\Delta(\mathsf{Activity3}) = \langle \downarrow: \{\mathsf{Activity2}\}, \mathsf{Begin}: \{\mathsf{uriA}\}, \mathsf{End}: \{\}\rangle$
$\Delta(\mathsf{uriB}) = \{\mathsf{uriA}\}$

Clearly there is a flow from uriA to uriB, since the set $\Delta(\mathsf{uriB})$ includes uriA. In addition, the set $\Delta(\mathsf{Activity3}).\!\downarrow$ includes Activity2 (which makes the activity stack evident), and the sets $\Delta(\mathsf{Activity3}).\mathsf{Begin}$ and $\Delta(\mathsf{Activity2}).\mathsf{End}$ include uriA, showing intermediate flows from uriA.

Similarly, example (iv) is well-constrained under the following hypotheses.

$\Delta(\mathsf{uriA}) = \{\}$
$\Delta(\mathsf{Activity6}) = \langle \downarrow: \{\}, \mathsf{Begin}: \{\}, \mathsf{End}: \{\}\rangle$
$\Delta(\mathsf{ACTION}) = \langle \mathsf{Send}: \{\mathsf{uriA}\}\rangle$
$\Delta(\mathsf{uriB}) = \{\mathsf{uriA}\}$

Again, clearly there is a flow from uriA to uriB, and the other sets encode a possible path for that flow.

4.5 Soundness of our analysis

We prove that our analysis is sound by reduction to a type system proposed in [7]. The soundness of that type system immediately implies the soundness of our analysis.

Theorem 4.1. *If an environment is data flow secure, then it is well-typed according to [7].*

Specifically, the key property of that type system is that well-typed environments preserve end-to-end secrecy and integrity of data: a value can flow from store m to store n only if readers of n may already read from m, and writers of m may already write to n. By the above theorem, any environment that we consider data flow secure by our analysis enjoys the same property.

The correspondence between the two analyses is not a coincidence. Indeed, our flow judgments may be viewed as type inference judgments, in the following sense. In [7], values are assigned types of the form $\mathsf{Any}(\mathtt{READ}, \mathtt{WRITE})$, meaning that such values may flow from contexts with at most permission WRITE to contexts with at least permission READ. Such types are related by a subtyping rule (where Γ is a set of type hypotheses, analogous to Δ):

$$\textbf{(Sub any)}\quad \frac{\mathtt{READ}' \geq \mathtt{READ} \qquad \mathtt{WRITE} \geq \mathtt{WRITE}'}{\Gamma \vdash \mathsf{Any}(\mathtt{READ}, \mathtt{WRITE}) <: \mathsf{Any}(\mathtt{READ}', \mathtt{WRITE}')}$$

Instead of associating values with types, we associate them with sets of stores whose permissions encode the relevant types; and the subtyping rule is analogous to our rule involving permission constraints for data flow security.

This analogy is also evident in the modular, assume-guarantee style reasoning in our analysis, which is reminiscent of type systems. In particular, the types assigned to activities and listeners in [7] carry similar information as flow sets in our analysis. For example, the type of a listener carries the parameter type of its listener function (which is similar to our Send flow set). Similarly, the type of an activity carries the parameter types of its onCreate and onActivityResult functions (which are similar to our Begin and End flow sets); in addition, it also carries the type of results returned by that activity, and has a special rule to connect stacks of activities (all of which are encoded conveniently by our $\downarrow$ flow sets).

Significantly, our analysis requires absolutely no annotations; in contrast, the type system in [7] requires type annotations for each component. In essence, our analysis is able to infer all of those annotations by generating the required constraints and solving them.

5 Implementation

We have implemented our analysis using WALA, a collection of open-source libraries for Java code analysis [6]. While the formal language of the previous section conveniently introduces several abstractions to simplify the analysis, applying the analysis to actual bytecode presents some significant challenges. For instance, we need a string analysis to recover addresses of components; we need a pointer analysis to track flows through the heap; we need to handle interprocedural flows through JVML bytecode; and so on. In this section we review some of the utilities provided

by WALA and describe the various elements of our analysis that we construct using these utilities.

5.1 Background on WALA

WALA provides a simple interface to parse a set of Java classes and generate a *call graph* over all reachable methods. Each node of the call graph is the *control flow graph* of some method. In addition, WALA provides interfaces to *explode* the call graph into an interprocedural "supergraph" over basic blocks of instructions.

Furthermore, WALA provides implementations of several standard program analysis frameworks. In particular, it supplies fixpoint solvers for *intraprocedural* data flow analysis and for *interprocedural* data flow analysis based on the RHS (Reps-Horwitz-Sagiv) algorithm [13]. Such problems can be set up by specifying the appropriate transfer functions across nodes and edges.

Finally, WALA implements a flexible *pointer analysis* framework. A pointer analysis maps local variables, fields, and other pointers to sets of objects they may point to; these sets of objects are represented by *instance keys*. The precision of a pointer analysis depends on the level of context sensitivity used to disambiguate objects created at the same program location, and WALA provides interfaces to customize this level of context sensitivity as necessary.

5.2 Elements of our analysis

Our analysis is composed of several elements, as shown in Figure 3. By separating each element of our analysis, we leave open the future possibility of replacing those elements with improved ones, or reusing those elements for other analyses.

Overall, our inputs include the classes and manifests of a set of applications, and our output is a set of permission constraints $\geq$ that are induced by data flows through those applications, which must be satisfied for end-to-end secrecy and integrity guarantees. We assume that a policy is specified as a partial order over permissions. The default policy considers permissions to be sets of string identifiers, interpreting $\geq$ as the superset relation over such sets. The verification condition is simply that the constraints generated by the checker are always admitted by such a policy.

To keep our analysis tractable, we hide Android's application framework code from WALA. Thus, control flows across components are not immediately visible to WALA. Our solution is to implement a modular analysis of flows within each component, followed by a final phase that connects and analyzes flows across all components. This allows us to analyze applications incrementally as they are installed, and better reflects the reusability of components across applications.

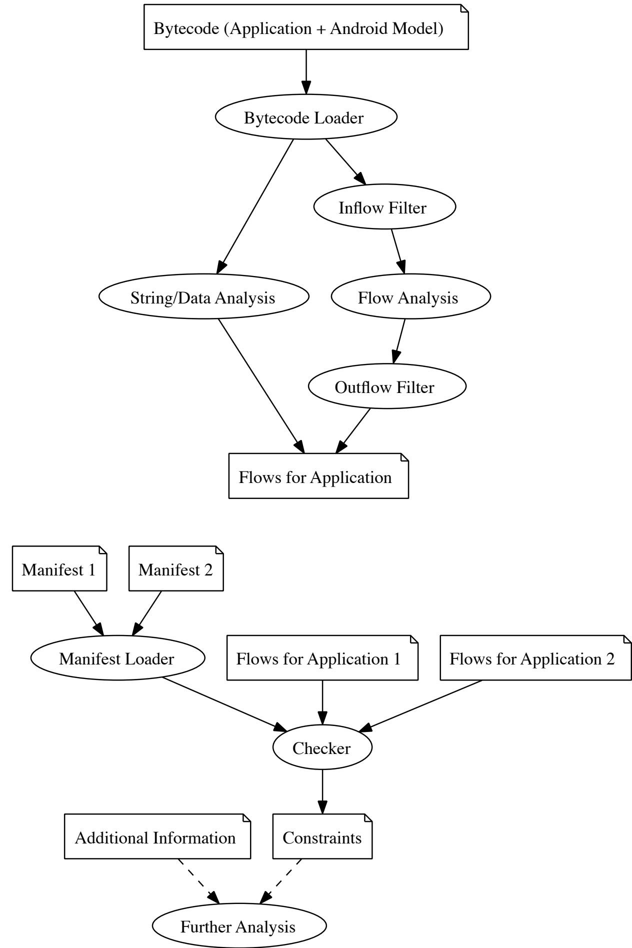

Figure 3. Architecture of analysis

Bytecode loader Our bytecode loader parses and generates a call graph from a given set of application classes and a fixed set of APIs exposed by Android. We retain only the code of the application, and prune away the APIs during analysis. Furthermore, instead of passing in the concrete implementations of the Android APIs, we pass in simple, abstract stubs—most of which come with the Android SDK, and some of which we extend with our own implementations where necessary. For example, we need to implement the APIs for manipulating Uri objects in android.net.Uri so that WALA's pointer analysis is able to account for objects created in and returned by such method calls. Similar implementations are required for code dealing with other Android data structures (such as Intent in android.content.Intent).

Furthermore, we customize the pointer analysis to add some necessary context-sensitivity for specific method calls. In particular, we force disambiguation of strings and other data structures returned by methods, based on information about their call sites, receivers, and so on. This pre-

cision becomes crucial in subsequent phases of our analysis.

Finally, for a modular analysis we consider multiple root methods, or *entry points*, for call graph generation. In particular, such entry points include callback methods for each component (*e.g.,* Activity.onCreate), callback methods that may be invoked by the UI thread (*e.g.,* Button.onClick), and all Runnable.run methods executed by threads that are launched within the application.

The outputs of the bytecode loader are a pruned call graph, an exploded interprocedural supergraph, and a pointer analysis.

String/data analysis Strings are used pervasively as *identifiers* in Android applications. For example, Uris are constructed from strings, and permissions are associated with strings in manifests. Strings are also used as query and update parameters, as keys into various data structures such as Bundles and ContentValues, and as class names or actions in Intents. Thus, precise tracking of data flows in Android requires a precise approximation of the values of various strings that appear in application code.

Technically, this is quite challenging. As we have seen in our examples, such strings may be stored in fields and variables and subjected to stateful operations (such as append). Thus, we must rely on a precise pointer analysis for strings, as well as partially evaluate any method calls that manipulate strings. In bytecode, strings are typically constructed and manipulated through StringBuilder objects, and coerced to and from such objects. We approximate the values of these objects as follows: first we construct a subgraph that focuses on operations over such objects, and next we design an analysis built on top of WALA's intraprocedural flow solver that uses this subgraph to compute approximate *prefixes* of the strings these objects may evaluate to at run time. Approximation is necessary because the pointer analysis may not be fully precise, and strings with imprecise values may be, for example, appended to StringBuilder objects.

This information is then propagated to other data structures (such as Uris) that may use those strings. Typically the strings we care about behave as constants, and it is possible to recover them fully (*i.e.*, the prefixes we compute are total). Otherwise, even incomplete prefixes can provide useful information: even if we cannot recover the parameter string in the suffix of a Uri, recovering the base Uri is sufficient to look up permissions for that Uri in a manifest.

The output of the string/data analysis is a map from instance keys for Strings and Uris (derived by the pointer analysis) to strings.

Inflow filter As described in section 2.4, data can only flow into an application through inflow methods. We treat each of the inflow methods in our application as a separate source. Each such source introduces an instance key in WALA, which we map to a unique *tag*. These tags contain enough information to solve inter-component flows, and are passed around by reference during the flow analysis. For example, we tag the Cursor returned by a call to query with an object that contains the Uri instance key that was passed to that query. A set of such tags corresponds to a flow set in our formal analysis (Section 4.1).

The output of the inflow filter is a map from instance keys (that identify sources) to tags. This map is used to seed our flow analysis.

Flow analysis Our flow analysis is best specified as an interprocedural flow problem, and we use WALA's built-in RHS solver in our implementation. Our specification includes a supergraph, a domain, and transfer functions that define how data flows between domain elements across edges in the supergraph.

In general, data may flow from instance keys and local variables to other instance keys and local variables. We include such instance keys and local variables in a set of *flow identifiers*. Flow identifiers correspond to values in our formal analysis (Section 4.1). Recall that unlike usual "taint" analysis, our analysis needs to simultaneously track flows of several tags for these flow identifiers. These tags are obtained from the output of the inflow filter. We define the domain used for our analysis to be the cross product of flow identifiers and tags. Thus, a set of domain elements with flow identifier v and some tag in F corresponds to a mapping $v \mapsto F$ in our formal analysis (Section 4.3).

The transfer functions are identical across tags, so we can define them simply over flow identifiers. For any instruction that is not a method call, let D be the set of local variables and instance keys that are *defined* and U be the set of the instance keys that are *used* in that instruction (these sets can be readily obtained through WALA's representation of the instruction with the general pointer analysis). Let B be the basic block containing this instruction, and *Pred*(B) be the predecessors of B in the supergraph. We specify a flow from each $u \in U$ in each $B' \in$ *Pred*(B) to each $d \in D$ in B.

Invoke instructions (*i.e.*, method calls) require special handling. A method call is either explicitly represented in the supergraph as a set of interprocedural edges, or it is an Android API call for which we have pruned the interprocedural edges. If the call is represented in the call graph we can simply specify flows of parameters across call edges and of results across return edges. However, when the call is not represented in the call graph we must consider all possible side effects of the invoke instruction at the location of the call. Android restricts side effects in its API calls to the objects that were passed to the call, so to be sound we

must add flows from each flow identifier used by the invoke instruction to each other flow identifier used or defined by that instruction. Note that for non-static methods, the uses of the instruction will also include a flow identifier representing the receiver of that call. For example, consider an instance v of some Android object, a non-static method p of the object, and a parameter v'. An instruction $x = v.p(v')$ would result in flows from (the flow identifiers for) v to (the flow identifiers for) x and v', and from v' to x and v.

There are a few points to note about our transfer functions. First, tracking flows through instance keys (derived by the pointer analysis) is crucial for an interprocedural analysis, since methods can have side effects through the heap that cannot be tracked by flows through local variables alone. Next, we focus only on explicit flows in our analysis, *i.e.*, we ignore implicit flows. Finally, our transfer functions specify a flow-sensitive analysis of local variables but a flow-insensitive analysis of heap variables (instance keys). This is important when considering multi-threaded applications, as we do not need to account for interleaving of multiple threads to achieve a sound analysis using this technique.

Given the supergraph constructed by the bytecode loader, the transfer functions as defined above, and the initial seeds provided by the inflow analysis, WALA's solver can calculate the domain elements that are active at each basic block. The result of the flow analysis is a map from flow identifiers to the tags they might have at each basic block. Thus, each entry in the map corresponds to a mapping $v \mapsto F$ in our formal analysis (Section 4).

Outflow filter Next, we identify sinks to which data may flow out of the application. Those sinks are the outflow methods described in section 2.4. The analysis in the outflow filter combines the set of all sinks in the application with the result of the flow analysis to derive all of the end-to-end, intra-component flows within the application. The output of the outflow filter is a map from (inflow) tags to sets of (outflow) tags. For example, the output for Activity1 would map the query inflow characterized by the instance key for uriA to the update outflow characterized by the instance key for uriB.

Manifest loader The manifest loader is responsible for characterizing the environment in which applications run, as described in section 2.2. Since our analysis must account for many different environments that are not fully defined at the time of analysis, we can only load a partial representation. In our manifest loader we typically load the manifests for the applications we are analyzing, coupled with manifests for core Android applications. Some of the characteristics of the manifests that our analysis requires include links between Uris and permissions, links between Uris and the ContentProviders that host those Uris, and the set of permissions that each application requests at install time. All such information is passed to the checker.

Checker The results of the outflow filter encode, in particular, flows between query and update calls that are represented by instance keys for Uri objects. We rely on our string analysis to recover approximate prefixes of these Uris, and we then look up the results of the manifest loader for any permissions known to be associated with these prefixes. Based on this information, we generate permission constraints $\geq$ that are induced by the flows.

We force the relation $\geq$ to be a partial order, *i.e.*, $\geq$ must remain reflexive, transitive, and anti-symmetric. This allows incremental checking as new applications are installed on an Android device; the system gets monotonically more constrained until we have a contradiction, which indicates a possible security violation. For example, suppose that a pre-installed application generates the constraint $p_x \geq p_y$ and a new application generates the constraint $p_y \geq p_z$. If as policy we know that $p_x \not\geq p_z$ (*i.e.*, p_z denotes an unrelated or strictly higher privilege than p_x), then we have a contradiction, which means that it is dangerous to let these applications coexist on the same Android device: either the former application should be uninstalled, or the latter application should not be installed.

Note that not all flows detected by our flow analysis may be permissible at run time. In particular, inter-component flows require that the relevant components have sufficient permissions to access each other. Thus, any constraints generated by the checker due to those flows are in fact *conditional* on such facts. For example, suppose that an overall flow from uri_A to uri_B is a composition of the following (intra-component) flows: within an activity in App_1, there is a flow from uri_A to an intent used to launch another activity in App_2; and within that activity in App_2, there is a flow from the intent used to launch it to uri_B.

Suppose that uri_A requires permissions $read_A$ and $write_A$, and uri_B requires permissions $read_B$ and $write_B$. Suppose further that (any activity in) App_1 has permission $perm_1$ and App_2 enforces, via its manifest, that calling any of its activities requires permission $perm_2$. Then the constraints that are generated by our checker are in fact:

- $perm_1 \geq perm_2 \Rightarrow read_B \geq read_A$ for data secrecy
- $perm_1 \geq perm_2 \Rightarrow write_A \geq write_B$ for data integrity

We expect these constraints to be useful for further analysis (which we do not implement). For instance, some interesting scenarios arise if either App_1 or App_2 is a new (not yet installed) application.

- Suppose that App_1 is a new application *requesting* some permission in its manifest. Rather than generate

constraints with $perm_1$ as that permission, we could assume $perm_1$ to be unknown and solve for an *optimal* solution. For example, an optimal solution may be to request the least privilege necessary to ensure functionality without compromising security.

- Similarly, suppose that App_2 is a new application *enforcing* some permission in its manifest. Again, rather than generate constraints with $perm_2$ as that permission, we could assume $perm_2$ to be unknown and solve for an optimal solution. For example, an optimal solution may be to enforce the greatest privilege sufficient to ensure security without compromising functionality.

5.3 Results

We have tested our analysis on all of the examples in Section 3, as well as on several other examples that exhibit longer chains of flows through other components and data structures, elided in this paper. The test cases we chose are designed to cover the space of inter-component Android flows. In each test case, we assigned permissions READ1 and WRITE1 to 'some.authority' and permissions READ2 and WRITE2 to 'some.other.authority'. Our analysis successfully solved for all such flows to find the constraints '[READ2] can read [READ1]' and '[WRITE1] can write [WRITE2]'. The time taken to analyze all of our simple test applications was only a few seconds, and we expect that our modular approach will scale nicely to typical Android application sizes and complexities. As we continue with this work we will be applying this analysis to real-world applications. Section 6 discusses some minor extensions that might be required to analyze such applications.

6 Limitations and future extensions

A slight current limitation of our analysis is that it requires either the Java source code or the compiled JVML bytecode of applications to be available. While this may be a reasonable model for offline certification (say, by the Android market), our analysis cannot be immediately applied to packaged applications on an Android device. This is because internally, Android works with a customized bytecode language for which there is (as yet) no known specification. Fortunately, there are ongoing efforts on building decompilers from that language to JVML [1] which, when available, can be plugged in as a front end to our analysis.

There are several ways of improving the precision of our analysis. The current modeling of API calls into Android's application framework is quite coarse; such calls can be modeled more accurately if we make our analysis aware of the semantics of those APIs. For example, we should be able to distinguish the flows of elements in a Bundle that are keyed with distinct strings, since we can already track such keys precisely with our string analysis. Furthermore, our current modeling of heap objects is flow insensitive, to conservatively tackle threading issues. We have not yet explored whether a thread-sensitive analysis will be necessary.

No matter how precise our analysis is, we expect that some of the flows we flag as security violations will in fact be "expected" flows. For example, a typical mail reader application will have permissions to read and write the user's contact list, as well as to send and receive messages across the Internet. Our analysis would likely find flows that exercise these capabilities in combination, and (without further information) flag those flows as security violations. In handling such an application, we have several options. We could either nominate the application for further (manual) review, or we could include some mechanism in our analysis to admit controlled downgrading [14] of flows through the relevant components. The latter may be desirable if we want to support automated security certification, and would require some lightweight annotations on the application that could be included in manifests alongside other declarations.

7 Related work

There is a huge body of work on program analysis for security; see [14] for a survey. We focus here on most closely related work.

Our basic approach is similar to that proposed in a recent study of language-based security on Android [7]. That paper studies a core formal language to describe Android applications, and proposes a type system to reason about end-to-end security of such applications. Our analysis tracks data flows rather than types; while types are useful for specifications, data flows seem to be more suited for implementations. Still, we are able to prove that our analysis essentially provides the same guarantees as that type system, by formalizing our core analysis in that language (Section 4). However, there are some significant differences between this work and [7]. First, [7] does not provide an implementation of their analysis; indeed, it does not even discuss any possible implementation issues. In contrast, we describe an implementation of our analysis that works on actual Android applications. Our implementation has to tackle significant challenges to justify the abstractions in the formal language. For instance, we need a string analysis to recover addresses of components; we need a pointer analysis to track flows through the heap; we need to handle interprocedural flows through JVML bytecode, and so on. Next, our analysis is fully automatic, and requires absolutely no annotations. On the other hand, the analysis proposed in [7] requires complicated type annotations for each component, which would impose an unreasonable burden on developers in an actual implementation. We find it remarkable that all

the information in these annotations can be inferred automatically by our analysis.

A related line of work [9, 8, 12] reports a logic-based tool, Kirin, for determining whether the permissions declared by an application satisfy a certain global safety policy. Typically, this policy bans "dangerous" combinations of permissions that may admit insecure data flows across applications. Despite this apparent similarity, however, our work takes a radically different approach from theirs, in the following ways. First, the policy enforced by Kirin is a "blacklist," meaning that any combination of permissions not specified as dangerous is considered fine. Unfortunately, applications may define and enforce their own permissions, and in general it seems impossible to specify a sound blacklist policy that eliminates all possible security violations. Next, a Kirin policy is typically subsumed by our general data flow security rule for generating permission constraints, *i.e.*, the permission combinations banned by such a policy can be usually derived as special cases of our permission constraints. Finally, Kirin does not track actual data flows through applications—it analyzes only manifests, not code. Thus, Kirin's analysis is necessarily less precise than ours. In particular, for an application that has several components, each of which require a disjoint set of permissions, Kirin conservatively considers the union of those permissions when deciding the safety of the application. In contrast, we track precise dependencies among the components, and thus may recognize the application to be safe even if Kirin cannot. This precision is important in the presence of signatures, which allow possibly unrelated applications to share the same set of permissions.

There are existing tools for verifying information-flow security of Java code, including, most notably, Jif [10]. We decided not to use Jif for the following reasons. First, Jif requires security type annotations throughout code, which we believe would force an unreasonable overhead for developers of Android applications. Furthermore, Jif is designed to enforce *noninterference*, which is a much stronger security guarantee than data-flow security. While noninterference is certainly attractive, we believe that its sensitivity to implicit flows makes it unsuitable as a general-purpose guarantee for Android applications.

Also closely related is TAJ, a tool for taint analysis in Java [15], which is primarily designed to verify web applications. Unlike that analysis, we need to track multi-commodity flows, *i.e.*, flows from multiple stores rather than a distinguished "taint source." Another important difference is that our analysis of applications is modular, and relies on an abstract semantics of applications rather than an analysis of Android's application framework code; this makes our analysis potentially more lightweight than theirs.

Two additional related projects are Mobius [4], supporting the checking of proof-carrying-code certificates on mobile devices, and S3MS [5], a framework for enforcing contract-based security for applications on mobile devices.

8 Conclusion

Android's access control mechanisms provide some basis for reasoning about the security of applications. However, without some sort of data flow analysis, the precision of reasoning by the user (and by extension any security reviewer) is necessarily limited. We provide a formal analysis for reasoning about data flows in Android applications. We also describe a technique for implementing data flow analysis on Java code for Android applications, and a tool that implements this technique. Although we have not yet applied our analysis to real-world applications, we have verified that our technique is sound on a set of applications that are representative of typical Android applications. We expect that our research will be particularly useful for providing automated security certification for Android applications.

References

[1] W. Enck. Personal communication.

[2] The Android "market" (application store). `http://www.android.com/market/`.

[3] The Android project (source code and SDK). `http://source.android.com/`.

[4] The Mobius project: Mobility, ubiquity and security. `http://mobius.inria.fr/twiki/bin/view/Mobius`.

[5] The S3MS project: Security of software and services for mobile systems. `http://www.s3ms.org/index.jsp`.

[6] WALA: Watson Libraries for Analysis (source code and plugins). `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[7] A. Chaudhuri. Language-based security on Android. In *PLAS'09: Programming Languages and Analysis for Security*, pages 1–7. ACM, 2009.

[8] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS '09: Computer and communications security*, pages 235–245. ACM, 2009.

[9] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security & Privacy Magazine*, 7(1):10–17, 2009.

[10] A. C. Myers, N. Nystorm, L. Zheng, and S. Zdancewic. Jif: Java + information flow. `http://www.cs.cornell.edu/jif`.

[11] G. C. Necula. Proof-carrying code. In *POPL'97: Principles of Programming Langauges*, pages 106–119. ACM, 1997.

[12] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC '09: Annual Computer Security Applications Conference*, 2009.

[13] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Principles of programming languages*, pages 49–61. ACM, 1995.

[14] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[15] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI '09: Programming language design and implementation*, pages 87–97. ACM, 2009.