Android Taint Flow Analysis for App Sets

William Klieber, Lori Flynn,
CERT / SEI, Carnegie Mellon University
{weklieber, lflynn}@cert.org

Amar Bhosale, Limin Jia, Lujo Bauer
Carnegie Mellon University
{amarb, liminjia, lbauer}@cmu.edu

Abstract

One approach to defending against malicious Android applications has been to analyze them to detect potential information leaks. This paper describes a new static taint analysis for Android that combines and augments the FlowDroid and Epicc analyses to precisely track both inter-component and intra-component data flow in a set of Android applications. The analysis takes place in two phases: given a set of applications, we first determine the data flows enabled individually by each application, and the conditions under which these are possible; we then build on these results to enumerate the potentially dangerous data flows enabled by the set of applications as a whole. This paper describes our analysis method, implementation, and experimental results.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.4.6 [*Security and Protection*]: Information flow controls

1. Introduction

Mobile devices such as smartphones and tablets are ubiquitous. The application environments on these devices implement a marketplace model, in which application developers publish apps which users can conveniently download and install from app stores. These apps can potentially access a variety of sensitive information, such as a user's location, contacts, and the unique identifier of the phone (IMEI). Applications such as social networking and banking apps can additionally collect and store a large amount of sensitive data. A significant concern in this setting is exfiltration of sensitive data, which may violate users' privacy and allow undesired tracking of users' behavior. Indeed, it has been shown that popular Android apps leak sensitive information, including location, IMEI number, phone number, and the SIM card ICC-ID [3].

Most mobile computing platforms, such as Android and iOS, use a permission system to attempt to limit the privileges of apps, including their ability to access and exfiltrate sensitive data. However, existing permission systems are not sufficient to prevent sensitive data from being leaked [3]. Additional analysis of data flows is thus necessary to determine whether sensitive data remains within expected boundaries, and to ensure that untrusted data does not contaminate a trusted data repository. Such analysis is often called *taint analysis*, and aims to determine whether data can flow from a sensitive data source to an undesired data sink. For instance, for a smartphone, data sources that contain sensitive data include the phone's unique identifier, SMS messages, etc., and apps that provide services such as banking. Undesired sinks for such data include network API or untrusted applications. We define a *source* as an external (to an app) resource from which data is read and a *sink* as an external resource to which data is written.

Taint analysis can be either static or dynamic. For instance, TaintDroid performs real-time taint tracking to dynamically detect data leaks [3]. In contrast, FlowDroid performs a highly precise taint flow static analysis for each component in an Android application [8, 9], and Epicc [14] analyzes properties of messages sent between components of apps. (Every Android app is composed of one or more components, as described in Section 2.) Advantages of static analyses include the absence of run-time overhead and the ability to detect harmful applications before they are even installed on a mobile device. However, there has been little work on statically analyzing dataflows of a system composed of several applications. This is important because data from a source might reach a sink only after passing through one or more components.

This paper describes our static taint analysis for Android, which can report undesired information flows that occur when several applications interact with each other. Our tool analyzes both inter-component and intra-component dataflow. It combines and augments the FlowDroid [9] and Epicc [14] analyses to precisely determine tainted flows both within and across applications. Our approach requires analysis of the source or bytecode of each app only once, and leverages the results to detect potentially dangerous flows enabled by all subsets of analyzed apps.

2. Background

Android Architecture. In Android, there are four types of app components: *activities*, which define a user interface; *services*, which perform background processing; *content providers*, which store and share data using a relational database; and *broadcast receivers*, which can receive broadcast messages from other applications. The primary method for inter-component communication, both within and between applications, is via *intents*. In this paper, we focus on communication between activity components. The `startActivity` family of methods is used to send an intent to an activity component. The app's manifest file specifies filters that are used by the system to determine if the app is eligible to receive a particular intent, using Android rules for matching filters to content of various intent fields. A component may also send an intent to itself, addressing itself either explicitly, or implicitly by setting its intent filter so that it can receive the intent it sent.

Static Analysis Tools. Our analysis is built upon the FlowDroid and Epicc analyses and the Soot analysis framework. The FlowDroid static analysis is context-, flow-, object-, and field-sensitive and Android app lifecycle-aware [9]. FlowDroid performs a highly precise taint flow static analysis for Android, but its analysis is lim-

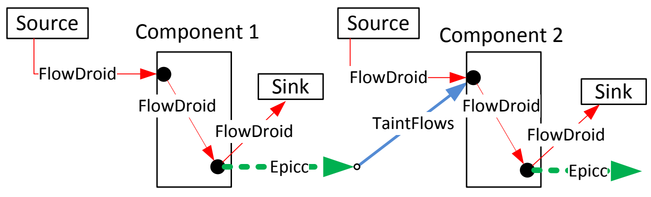

Figure 1. Analysis by data flow type: FlowDroid identifies sources (including intents received), flow of the data within the component, and sinks (including intents sent). Epicc identifies characteristics of intents sent by a component. TaintFlows, the Phase-2 analyzer, matches sent intents to components which could receive the intent, using app manifest data and matching intent IDs from Epicc and FlowDroid. From beginning to end, a given data flow can be internal to one component or traverse multiple components that can be in a single app or multiple apps.

ited to single components. FlowDroid's analysis uses a static list of Android API methods that correspond to sources and sinks. This list is produced by SuSi [15], a tool that uses machine learning to classify the methods exposed by the Android API.

The Epicc tool precisely and efficiently identifies properties (such as *action*, *category*, and *data MIME type*) of intents that can be sent and received by components [14]. For example, Epicc might identify that a particular app can only send intents with action `android.intent.action.VIEW` and MIME type `image/jpg`.

Soot [16] is a Java optimization and analysis framework. We use the Soot framework in several parts of our analyzer described in Section 4.

3. Analysis Method

The overview of our analysis method is shown in Figure 1. The goal of our analysis is to produce a set of all possible source-to-sink flows within a set of Android apps. Our taint flow analysis takes place in two phases. In Phase 1, each application is analyzed individually. Received intents are considered sources, and sent intents are considered sinks. The output of our Phase-1 analysis, for each app, consists of (1) flows within each component, found by FlowDroid; (2) identification of the properties of sent intents, as found by Epicc; and (3) intent filters of each component, extracted from the manifest file.

An intent ID is assigned to every source code location that sends an intent (i.e., a source code location that consists of a call to a method in the `startActivity` family), as described in Section 4.1.1. Sent intents with distinct IDs are considered distinct sinks, while intents with the same ID are combined together.

Phase 2 of the analysis is carried out on a particular set of apps, using the output of Phase 1. The output of Phase 2 consists of all the source-to-sink flows found in the set of apps.

3.1 Example Scenario

We next introduce an example of information flows between several components (Figure 2) that cannot be precisely analyzed by existing tools. Suppose that Component C_1 sends data to Component C_2 and receives data from it in return. Component C_3 interacts with C_2 in a similar fashion. We do not specify whether these three components belong to different apps or to a single app; the analysis is the same in either case. As depicted in Figures 2 and 3, for $i \in \{1, 3\}$:

1. Component C_i calls `startActivityForResult` to send data from source src_i to component C_2 via intent I_i.

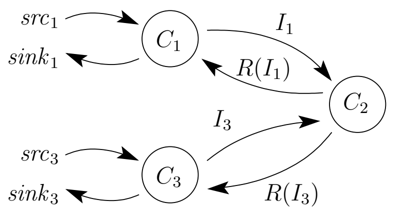

Figure 2. Running example described in Section 3.1. $R(I_i)$ denotes the response to intent I_i (set using `setResult`).

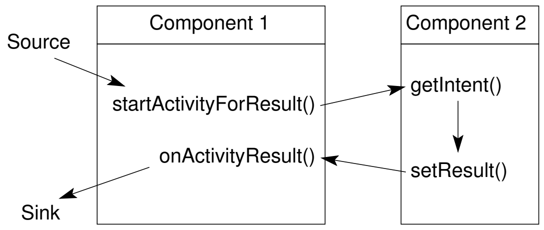

Figure 3. Interaction between C_1 and C_2 in the running example

2. Component C_2 reads data from intent I_i and sends that data back to component C_i by calling `setResult`.
3. Component C_i, in method `onActivityResult`, reads data from the result and writes it to sink $sink_i$.

To be precise, our analysis should determine that (1) information flows from src_1 to $sink_1$ (but not $sink_3$), and (2) information flows from src_3 to $sink_3$ (but not $sink_1$). Note that FlowDroid by itself cannot produce this precise of a result, even in the case where the three components are part of a single app.

3.2 Phase 1

In this phase, we analyze each app individually. We identify an intent by a tuple of (sending component, receiving component, intent ID). An intent sent from C_1 to C_2 with ID id will be denoted by $I(C_1, C_2, id)$.

In Phase 1, when a component calls a method in the `startActivity` family, we do not know which components can potentially receive the the intent (because each app is analyzed in isolation in Phase 1, but potential recipients can be components of other apps), so we use null for the recipient field. Likewise, in the `onCreate` method (which the Android OS calls when the activity component receives an intent), we do not know the sender of the intent, so we use null for the sender field. If a component receives an intent I_1 and returns information via the `setResult` method, we denote the returned information by $R(I_1)$.

We write $source \xrightarrow{C} sink$ to denote that information flows from $source$ to $sink$ in component C. For this purpose, we treat intents as both sources (in the component that creates and sends the intent) and sinks (in the component that receives the intent). Using this notation, we represent the flows depicted in Figure 2 and described in Section 3.1 as follows:

$$\begin{aligned} src_1 &\xrightarrow{C_1} I(C_1, \text{null}, id_1) \\ R(I(C_1, \text{null}, \text{null})) &\xrightarrow{C_1} sink_1 \\ I(\text{null}, C_2, \text{null}) &\xrightarrow{C_2} R(I(\text{null}, C_2, \text{null})) \\ src_3 &\xrightarrow{C_3} I(C_3, \text{null}, id_3) \\ R(I(C_3, \text{null}, \text{null})) &\xrightarrow{C_3} sink_3 \end{aligned}$$

The above flows constitute the desired output of the FlowDroid analysis. Although all the flows in the running example involve intents, in general our analysis will also find flows from non-intent sources to non-intent sinks.

We focus, in both description and in implementation, on intents sent and received by `Activity` components; other types of components (services, content providers, broadcast receivers) can be handled similarly.

3.3 Phase 2

After all apps in the given set have been analyzed, we enter Phase 2. Our goal is to find out how tainted information can flow between components. For each sent intent, we find all possible recipients, and we instantiate the Phase-1 flow equations (which have missing sender/receiver information) for all possible sender/receiver pairs, as we describe in detail in Section 3.3.1. For the running example, the Phase-2 flow equations are as follows:

$$\begin{aligned} \mathit{src}_1 &\xrightarrow{C_1} I(C_1, C_2, \mathit{id}_1) \\ R(I(C_1, C_2, \mathit{id}_1)) &\xrightarrow{C_1} \mathit{sink}_1 \\ I(C_1, C_2, \mathit{id}_1) &\xrightarrow{C_2} R(I(C_1, C_2, \mathit{id}_1)) \\ I(C_3, C_2, \mathit{id}_3) &\xrightarrow{C_2} R(I(C_3, C_2, \mathit{id}_3)) \\ \mathit{src}_3 &\xrightarrow{C_3} I(C_3, C_2, \mathit{id}_3) \\ R(I(C_3, C_2, \mathit{id}_3)) &\xrightarrow{C_3} \mathit{sink}_3 \end{aligned}$$

Let $T(s)$ denote the taint of s, that is, the set of sensitive sources from which s potentially has information. The goal of the analysis is to determine the taint of all sinks. Each phase-2 flow equation $s_1 \to s_2$ relates the taint of s_1 to the taint of s_2. If data flows from s_1 to s_2, then s_2 must be at least as tainted as s_1. Accordingly, we generate a taint equation $T(s_1) \subseteq T(s_2)$. For the running examples, the taint equations we generate are:

$$\begin{aligned} T(\mathit{src}_1) &\subseteq T(I(C_1, C_2, \mathit{id}_1)) \\ T(R(I(C_1, C_2, \mathit{id}_1))) &\subseteq T(\mathit{sink}_1) \\ T(I(C_1, C_2, \mathit{id}_1)) &\subseteq T(R(I(C_1, C_2, \mathit{id}_1))) \\ T(I(C_3, C_2, \mathit{id}_1)) &\subseteq T(R(I(C_3, C_2, \mathit{id}_3))) \\ T(\mathit{src}_3) &\subseteq T(I(C_3, C_2, \mathit{id}_3)) \\ T(R(I(C_3, C_2, \mathit{id}_3))) &\subseteq T(\mathit{sink}_3) \end{aligned}$$

Each non-intent source s is tainted with itself, i.e., $T(s) = \{s\}$. We then find the least fixed-point of the set of taint equations. The end result of Phase 2 is the set of possible source to sink flows.

3.3.1 Details of Generating Phase 2 Flow Equations

Let S be the set of sources and sinks (including intents and intent results) in the Phase 1 flow equations. Consider a transmitted intent I_{TX} and a received intent I_{RX} from Phase 1. In all cases, I_{TX} will have the form $I(C_{TX}, \mathsf{null}, \mathit{id})$, and I_{RX} will have the form $I(\mathsf{null}, C_{RX}, \mathsf{null})$.

In Section 3.3, we said that we instantiate the Phase-1 flow equations for all possible intent sender/receiver pairs. We now give the details of how we do this. For each Phase-1 flow $\mathit{src} \to \mathit{sink}$, we generate the set of all flows of the form $\mathit{src}' \to \mathit{sink}'$ that satisfy the following conditions:

1. If *src* is a regular (non-intent) source, then $\mathit{src}' = \mathit{src}$.
2. If *sink* is a regular (non-intent) sink, then $\mathit{sink}' = \mathit{sink}$.
3. If *src* has the form $I(\mathsf{null}, C_{RX}, \mathsf{null})$ (for example, the result of a call to `android.app.Activity.getIntent()`), then src' must have the form $I(C_{TX}, C_{RX}, \mathit{id})$ where there exists an intent $I(C_{TX}, \mathsf{null}, \mathit{id}) \in S$ that matches the intent filter of component C_{RX}.
4. If *sink* has the form $I(C_{TX}, \mathsf{null}, \mathit{id})$ (for example, an intent object passed to `startActivity`), then sink' must have the form $I(C_{TX}, C_{RX}, \mathit{id})$ where component C_{RX} has an intent filter that matches the intent *sink*.
5. If *src* has the form $R(I(C_{TX}, \mathsf{null}, \mathsf{null}))$ (for example, a parameter of the callback method `onActivityResult`), then src' must have the form $R(I(C_{TX}, C_{RX}, \mathit{id}))$ where
   (a) there exists an intent $I(C_{TX}, \mathsf{null}, \mathit{id}) \in S$ that matches the intent filter of component C_{RX}, and
   (b) $R(I(\mathsf{null}, C_{RX}, \mathsf{null})) \in S$.
6. If *sink* has the form $R(I(\mathsf{null}, C_{RX}, \mathsf{null}))$ (for example, a value passed to `setResult`), then sink' must have the form $R(I(C_{TX}, C_{RX}, \mathit{id}))$ where
   (a) there exists an intent $I(C_{TX}, \mathsf{null}, \mathit{id}) \in S$ that matches the intent filter of C_{RX}, and
   (b) $R(I(C_{TX}, \mathsf{null}, \mathsf{null})) \in S$.
7. If *src* has the form $I(\mathsf{null}, C_{RX}, \mathsf{null})$ and *sink* has the form $R(I(\mathsf{null}, C_{RX}, \mathsf{null}))$, then sink' must be $R(\mathit{src}')$.

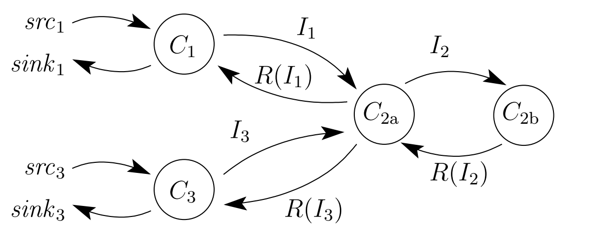

Figure 4. Example of inter-app communication flow wherein, for $i \in \{1, 3\}$: $C_{2\mathrm{a}}$ receives tainted data from C_i, sends it to $C_{2\mathrm{b}}$, receives a result with the same taint, and finally sends it back to C_i.

Condition 7 allows us to precisely handle a situation in which a component (such as C_2 in the running example) processes data from various callers without intermingling the taintedness of the data. Condition 7 is sound as long as multiple instances of the component can communicate only via flows included in the Phase-1 equations. Our current implementation catches most such flows but misses inter-instance communication via static fields.

For example, in Figure 2, if all components are part of the same app, then the two launched instances of C_2 can store information from I_1 and I_3 in a static field (which is shared between the two instances of C_2). The value in the static field (tainted with both src_1 and src_3) can then be read and copied into $R(I_1)$ and $R(I_3)$. This flow would be missed by our current analysis.

For future work, we plan to address static fields in a sound manner. In particular, if an app A has a class C with a static field sf, we would modify FlowDroid to introduce a dummy entity $\mathit{sf}_{A,C}$ that can act both as a source and as a sink. Reading from static field sf would be treated as reading from $\mathit{sf}_{A,C}$, and writing to sf would be treated as writing to $\mathit{sf}_{A,C}$. The resulting Phase-1 flow equations would enable our Phase-2 analysis to soundly handle inter-instances communication via static fields.

Our analysis cannot precisely handle the situation in Figure 4, wherein tainted data travels through a chain of apps. In this situation, our analysis would mark all intent results as being tainted with data from both I_1 and I_3 instead of being able to keep them separate.

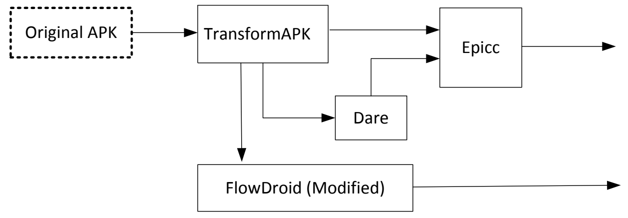

Figure 5. Phase 1

3.3.2 Rules for Matching Intents

In Section 3.3.1, we used the term "match" in relation to a sent intent and an intent filter. We now more fully define what we mean by "match". The Android documention[1] describes how a sent intent is matched to potential recipients. If the intent explicitly designates a recipient, then the intent is matched with exactly that recipient. Otherwise, the intent is matched with a filter if it passes three tests: an action test, a category test, and a data test.

Epicc provides information about outgoing intents in its app analysis, and we use that. It provides no information about the URI fields of intents, so we ignore the URI fields when matching intents with intent filters.

Sometimes, Epicc will return `<any_string>` for the action string or `Found top element` for the intent as a whole. For this case, the analyzer has two modes (which can be selected by a command-line option): (1) a sound mode, which assumes that an unknown action string potentially matches any action string in any filter, thereby typically generating many false positives, and (2) a precise mode, which assumes that the unknown action string doesn't match any filter, thereby potentially generating false negatives. Likewise, in the sound mode, a top-element intent matches every filter, and in the precise mode, it matches nothing.

4. Implementation Details

We have implemented our approach in an analyzer that we call "DidFail" (Droid Intent Data Flow Analysis for Information Leakage). Our analyzer (source code and binaries), along with 3 apps which demonstrate the running example in §3.1, is available at: `http://www.cert.org/secure-coding/tools/didfail.cfm`

4.1 Phase 1 Analysis

Figure 5 show the components of our analyzer, the processing sequences, and dataflow paths. The analyzer incorporates use of the previously existing and unchanged tools Epicc, Dare, and Soot; a modified version of FlowDroid; and new tools TransformAPK and TaintFlows. TaintFlows performs the Phase 2 analysis.

4.1.1 APK Transformer

The APK Transformer must be used in the first step of the analysis, in order to be able to integrate results of the different analytical tools used afterwards. This step was critical in order to achieve our ultimate goal of outputting detected paths from sources to sinks, including paths which contain dataflows via intents. Android apps are packaged in files with the extension *.apk*. In Figure 5, "Original APK" is the original Android app. With the APK Transformer, our analyzer modifies that app to enable matching intents mentioned in both the Epicc and FlowDroid outputs. To do this, we transformed each original .apk file into a modified .apk file, using Soot. We wrote a program that first uses Soot to transform the .dex Android bytecode into an intermediate representation called *jimple*. The program uses the Soot framework to locate method calls that send intents, and immediately before that, we insert new jimple code, which calls an Android method that inserts a unique ID into the intent. Then, our program uses Soot to compile the modified jimple code into a new .apk file. When Epicc processes this modified file, it prints the unique intent IDs. As described in Section 4.1.2, we modified the source code of FlowDroid so that its output identifies sent intents by their intent ID, enabling us to match intent analysis from the two tools. We could not modify the source code of Epicc, because it is currently published only as a .jar file. According to the Epicc website, the authors plan to publish the source code in the future. After the source code is available, we might be able to combine FlowDroid and Epicc in a more efficient manner.

4.1.2 FlowDroid (Modified)

We modified FlowDroid in several ways. For our Phase-1 analysis, we consider the points where intent information flows in and out of the component as sources and sinks respectively. Therefore, we added the method `onActivityResult()` as a source and `setResult()` as a sink in FlowDroid. The methods `getIntent()` and `startActivityForResult()` were already present as a source and sink respectively. Although the FlowDroid tool comes with a smaller `SourcesAndSinks.txt` file, one can substitute the much larger `SourcesAndSinks.txt` file from the SuSi analyzer [15][2]. We also added code to look for the `putExtra` call we added to insert the unique intent ID, which was added by the APK Transformer. In flows where an intent is the sink, the output of FlowDroid identifies the intent by its unique ID.

In a flow $src \xrightarrow{C} I(C, \mathsf{null}, id)$, how do we identify C? When an intent is sent via `base.startActivity` (including the case where `base` is an implicit `this`), we assume that that the class of `base` must be the sending component.[3]

The output of FlowDroid was originally non-deterministic in the order in which flows were listed. To produce deterministic output for regression testing, we simply sorted the flows before printing.

4.1.3 Epicc and Dare

The Dare [13] tool takes the transformed .apk file as input, retargets the application, and outputs Java class files. The Epicc analysis takes two inputs: the transformed .apk file and the output of Dare.

4.2 Phase 2 Analysis

Each app in the app set undergoes its own separate Phase-1 analysis, with each Phase-1 analysis output providing three separate output files (manifest file, Epicc output and FlowDroid output) that are input to the Phase-2 analysis. If there are n apps in the app set, then the Phase-1 analysis is performed n times, outputting $3n$ files, all of which are input to the (single) Phase-2 analysis. The Phase-2 analysis output provides information about dataflows from a source to a sink, including intents if they are part of the data flow.

5. Experimental Results

We tested our prototype analyzer on two app sets. App Set 1 contains 3 apps that we created, which match the running example in Figure 2. App Set 2 contains 3 apps from the DroidBench benchmark suite [2] that use intents for inter-app communication.

[1] `http://developer.android.com/guide/components/intents-filters.html#Resolution`

[2] https://github.com/secure-software-engineering/SuSi (2013-11-25)

[3] We have not yet been able to confirm or refute whether this is a sound assumption. To preserve soundness at the expense of precision, we considered an intent as potentially matching the intent filter of all components of an app if it matches the intent filter of any component of the app.

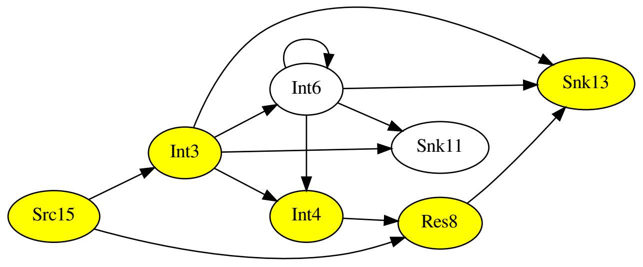

Figure 6. Tainted data flow in DroidBench apps. Graph created by running Graphviz on output of our analyzer. Src15=`getDeviceId`, Int3=I(IntentSink2,IntentSource1,3), Int4=I(IntentSource1,IntentSink1,4), Res8=R(Int4), Snk13=`Log.i`

Our analyzer successfully traced all inter-app and intra-app flows in both app sets. As described in the previous section, we first ran the Phase 1 analysis on all apps individually, and then we ran the Phase 2 analysis for each set of apps.

App Set 1

- SendSMS.apk: This app leaks the user's DeviceId through an SMS. It reads the user's DeviceId, then adds it to an intent using the `putExtra` method. It then sends this intent out by calling `startActivityForResult`. Another app receives this intent and responds with a result. When the intent result is received, the `onActivityResult` callback method is called. Data received in the result is then leaked through an SMS.
- Echoer.apk: This app receives intents from other apps. It reads the incoming intents using the `getIntent` method, and writes the received data using `Log`. Also, it sends this data back to the transmitter using the `setResult` call.
- WriteFile.apk: This app is similar to SendSMS except that it reads the user's location and leaks it to the FileSystem.

Result: Our analysis detected the following interesting flows:

▪ $getDeviceId \xrightarrow{SendSMS} startActivityForResult$
$getIntent \xrightarrow{Echoer} setResult$
$onActivityResult \xrightarrow{SendSMS} sendTextMessage$

▪ $getLastKnownLocation \xrightarrow{WriteFile} startActivityForResult$
$getIntent \xrightarrow{Echoer} setResult$
$onActivityResult \xrightarrow{WriteFile} write$

App Set 2: Droidbench Benchmark Suite DroidBench is a set of open source apps for testing static analysis tools.

- IntentSource1.apk: This app reads the incoming intent using `getIntent`, and sends the intent out by calling `startActivityForResult`. When another app receives this intent and responds with a result, this app logs the result (Sink).
- InterAppCommunication_IntentSink1.apk: This app read the user's DeviceId (Source), adds it to the received intent, and then sends the intent out by calling `getIntent` method.
- InterAppCommunication_IntentSink2.apk: This app also reads the user's DeviceId (Source), adds it to a new Intent object, and sends the intent out by calling `startActivty`.

Result: Our analysis detected the following interesting data flows as shown in the Figure 6.

▪ $Src15 \xrightarrow{IntentSource1} Int3 \xrightarrow{IntentSource1} Snk13$

▪ $Src15 \xrightarrow{IntentSource1} Res8 \xrightarrow{IntentSource1} Snk13$

Note: The apps InterAppCommunication_IntentSink1 and InterAppCommunication_IntentSink2 use the same package name 'de.ecspride'. Since Android does not allow two packages with the same name, we changed the package names to 'de.ecspride.IntentSink1' and 'de.ecspride.IntentSink2'.

6. Sources of Unsoundness and Imprecision

Among its sources of unsoundness and imprecision, our analysis inherits those from its building blocks, Epicc and FlowDroid.

Unsoundness Sources of unsoundness cause the analysis to fail to identify a tainted flow. Sources of unsoundness in our analysis include reflection and native code, which are not addressed by Epicc. FlowDroid also does not consider reflective calls. However, FlowDroid does analyze calls that invoke native code, using a heuristic called *Taint Wrapping*. It defines explicit taint propagation rules for commonly called native methods. For all other native methods, FlowDroid uses the following heuristic: if the input array was tainted before the call, then FlowDroid determines that all call arguments and any return value are tainted. FlowDroid's handling of native calls is not sound; it does not analyze the native code in the callee. For example, native code can read from sources and write to sinks, which will not be detected by FlowDroid. FlowDroid also is unsound because it does not trace some leaks caused by multi-threading and some implicit flows. If the component's life cycle is not modeled completely by Epicc and/or FlowDroid, that would be another source of unsoundness, as discussed in FlowDroid [9] and Epicc [14] papers.

The analysis that we have presented does not consider implicit flows that involve the mere receipt of intents without reading any information from the received intents. For example, suppose an app A_T wants to communicate a bit vector $\langle b_n, ..., b_0\rangle$ to an app A_R without being detected by our analysis. App A_R can have two components, C_{R0} and C_{R1}, which have mutually exclusive intent filters. Then A_T can send a sequence of intents $\langle I_n, ..., I_0\rangle$ where

- intent I_i matches C_{R0} iff $b_i = 0$, and
- intent I_i matches C_{R1} iff $b_i = 1$.

To ensure that intents arrive in proper order, App A_T can use `startActivityForResult` to send the intent and then wait until C_{R0} or C_{R1} calls `setResult` to acknowledges receipt.

Data can flow between components of different apps via file or database accesses such as writes to and reads from shared external storage, internal storage, and shared public directories on the device. These same file and databases can be accessed to allow (and sometimes to restrict) data flow between components of a single app. FlowDroid considers a read from a file to be a source and a write to a file to be a sink. Within one component, the FlowDroid analysis finds a tainted flow if there is a read from a file and a call to a sink, or a call to a source and a write to a file. Although our analyzer finds some tainted flows, including file access, which FlowDroid does not, it does not soundly analyze taint flows involving files accesses. Our analysis finds a multi-component tainted dataflow that ends with a write to a file sink. However, it does not trace a multicomponent tainted dataflow that starts with a read from a file source. Also, our analyzer is unsound because it does not trace a multi-component tainted dataflow with a read from a file in one component after a write to it by another.

Additional sources of unsoundness in the analyzer include shared static fields. FlowDroid traces tainted data within a component (or within an entire app, depending on command line arguments) that is written to and/or read from a shared static field. Unsoundness due to inter-instance static field communication is discussed in Section 3.3.1.

Imprecision Imprecision in the analysis would result in the analysis reporting a possible tainted flow where such flow is not actually possible. For instance, the Epicc analyzer over-approximates inter-component communication (ICC) because it does not handle URIs, which are used by Android to match intents to receiving components. As described above, FlowDroid's analysis of native calls is not precise and sometimes over-approximates returned tainted fields. Our analyzer does not use permissions to restrict possible matching of intent senders and receivers. This over-approximated matching is a source of analysis imprecision.

In our Phase-1 FlowDroid analysis, all the received intents for a component are conflated together as a single source. As future work, to be more precise, we plan to modify FlowDroid so that, when a callback function such as `onCreate` is analyzed, it can report the data flows as a function of the properties of the received intent. For example, we might report that a component C has a flow $camera \xrightarrow{C} R(I)$ iff $I.hasExtra(\text{“}\texttt{cam}\text{”}) = \texttt{true}$. Similarly, we can make analysis of `onActivityResult` be sensitive to the value of the `requestCode` parameter.

7. Related Work

The Epicc tool performs the most precise static analysis of Android intents of any Android analyzer known to us, finding vulnerabilities with far fewer false positives than the next best tools. The authors showed that the intent ICC problem can be reduced to an inter-procedural Distributive Environment (IDE) problem, so the existing algorithms for efficient IDE solutions could be used. Epicc builds on a pre-existing IDE framework within the Soot library.

Daniel Hausknecht's 2013 thesis [10] describes VarDroid, intended to integrate intra-component and inter-component static analyses, similar in some ways to our method. His concept is modular where different analyses could be switched out for the intra-component and inter-component dataflow tracing. Where we use a modified FlowDroid, his concept could use Chex [12], FlowDroid, or another analyzer. His thesis says he did not complete integrating FlowDroid in his system. Instead, he simulated dataflow analysis through probabilistically generated information simulating results of the intra-component and inter-component analyses.

The Kirin tool [4] provides a formalized model for stating data policy and compares stated policies to information extracted from app manifest files, processing this information on the phone, to determine whether an app should be installed. The SORBET [7] system modified a standard Android system to enable formal definition of desired security properties, which were proven to hold on SORBET but not on Android. Livshitz et al. [11] did static analyses on Java code to detect policy violations with security implications, including taint analysis. TaintDroid [3] does realtime taint tracking to dynamically detect data leaks.

Felt et al. [5] found that about one-third of Android apps (of the 940 they tested) asked for more privileges than actually used. They found evidence that a cause of over-privilege is developer confusion in part due to insufficient Android API documentation. Furthermore, malicious apps can use permission re-delegation attack methods [6], which when successful take advantage of a higher-privilege app performing a privileged task for an application without permissions. The ComDroid [1] tool analyzes inter-app communication in Android, looking at intents sent and the manifest files for potential vulnerabilities due to app communication. Although it examines vulnerabilities due to intents, the ComDroid analysis does not trace and identify data paths between sources and sinks.

8. Conclusions and Future Work

This paper introduced a new analysis that integrates and enhances existing Android app static analyses in a two-phase method. We demonstrated feasibility by implementing our approach and testing apps with it. Future work planned includes enhancing the inter-component part of the taint flow analysis to include additional data channels such as static fields, SQLite databases, and SharedPreferences. We also plan to test a large number of publicly available Android apps. We envision that a two-phase analysis such as ours can be used as follows. An app store can run the Phase-1 analysis on each of the apps in the app store. When a user wants to install a new app, the app store would conduct the Phase-2 analysis and tell the user about the new flows that would be made possible if the new app is installed.

Acknowledgments We thank Dr. Bodden for encouraging us to work on this analysis that incorporates Epicc and FlowDroid analyses. We thank Robert Seacord for useful discussions and the anonymous reviewers for their comments. This work was funded[4] by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon (for the operation of the SEI, an FFRDC), by ONR grant N000141310156, and by NSF grant CNS-1320470.

References

[1] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. MobiSys*, 2011.

[2] ECSPRIDE. DroidBench Benchmarks. Accessed 03-26-2014.

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. OSDI*, 2010.

[4] W. Enck, M. Ongtang, and P. D. McDaniel. Understanding Android Security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. CCS*, 2011.

[6] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security*, 2011.

[7] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and enhancing Android's permission system. In *Proc. ESORICS*. 2012.

[8] C. Fritz. FlowDroid: A Precise and Scalable Data Flow Analysis for Android. Master's thesis, TU Darmstadt, July 2013.

[9] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. PLDI*, 2014. To appear.

[10] D. Hausknecht. Variability-aware Data-flow Analysis for Smartphone Applications. Master's thesis, TU Darmstadt, Sept. 2013.

[11] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. USENIX Security*, 2005.

[12] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*, 2012.

[13] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android applications to Java bytecode. In *Proc. FSE*, 2012.

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

[15] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proc. NDSS*, 2014.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - A Java bytecode optimization framework. In *Proc. CASCON*, 1999.

[4] Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense or other sponsors. This material has been approved for public release and unlimited distribution.