SDLI: Static Detection of Leaks across Intents

Rocco Salvia
School of Computing
University of Utah
Salt Lake City, Utah, USA
rocco@cs.utah.edu

Pietro Ferrara
JuliaSoft SRL
Verona, Italy
pietro.ferrara@juliasoft.com

Fausto Spoto
Università di Verona
and JuliaSoft SRL
Verona, Italy
fausto.spoto@univr.it

Agostino Cortesi
DAIS
Università Ca' Foscari
Venice, Italy
cortesi@unive.it

***Abstract*—Intents are Android's intra and inter-application communication mechanism. They specify an action to perform, with extra data, and are sent to a receiver component or broadcast to many components. Components, in the same or in a distinct app, receive the intent if they are available to perform the desired action. Hence, a sound static analyzer must be aware of information flows through intents. That can be achieved by considering intents as both source (when reading) and sink (when writing) of confidential data. But this is overly conservative if the intent stays inside the same app or if the set of apps installed on the device is known in advance. In such cases, a sound approximation of the flow of intents leads to a more precise analysis. This work describes SDLI, a novel static analyzer that, for each app, creates an XML summary file reporting a description of the tainted information in outwards intents and of the intents the app is available to serve. SDLI discovers confidential information leaks when two apps communicate, by matching their XML summaries, looking for tainted outwards intents of the first app that can be inwards intents of the second app. The tool is implemented inside Julia, an industrial static analyzer. On some popular apps from the Google Play marketplace, it spots inter-apps leaks of confidential data, hence showing its practical effectiveness.**

***Index Terms*—Information Flow, Static Analysis, Abstract Interpretation, Android, Intent.**

I. Introduction

This article reports design and implementation of a novel intent analysis that detects sensitive information flows across different Android applications (*apps*) and its integration inside the Julia static analyzer. Our analysis warns when an intent carries sensitive data from one app to another. Hence, it tracks intent flows both between components and between apps. It uses Julia's flow analysis for tracking flows of tainted data, using an abstraction in terms of Boolean formulas to express all possible explicit flows of tainted data [5]. Starting from a set of sources of tainted user input, it establishes if tainted data flows into sensitive sinks.

SDLI is the implementation of such analysis, on top of the Julia analyzer. For each app, SDLI computes a soundy XML summary file reporting a description of the tainted information in outwards intents and of the intents the app is available to serve. By matching pairs of XML files, one can discover leaks of confidential information when two apps communicate, by checking if an output tainted intent can pass the app's boundaries and reach another app.

The main novelty of our analysis is that it considers all kinds of Android's components and both implicit and explicit intents. Moreover, it has been widely evaluated over a large set of real world apps from the Google Play marketplace, where its ability to deal with implicit intents uncovers dangerous flows. Finally, the analysis is soundy [9], since it inherits the power and precision but also the limitations of Julia [17], [10] and of most static tools that analyze real software (typically because of reflection and multithreading).

This article is organized as follows. Sec. II discusses the state of the art on intent analysis. Sec. III presents the structure of Android apps and the use of intents. Sec. IV recalls Julia's flow analysis and shows its application to Android. Sec. V presents the novel intent analysis. Sec. VI reports experiments. Sec. VII concludes.

II. Related Work

Most analyses dealing with intents focus on a limited set of components only, typically activities. Some analyze the Android Dalvik (.dex) bytecode directly, others first translate it into Java bytecode or other intermediate language, then perform the analysis [16]. We now compare our approach to some of the most notable tools.

FlowDroid [2] performs taint analysis on .dex files. It collects sources and sinks using SuSi [15] and it considers any method that sends intents as a sink and any call that extracts data from an intent as a source. It does not deal with implicit intents nor with intent modifiers, unless they are constant strings. Epicc [13] performs a sound static analysis of Android apps in Java bytecode, translated from .dex code through Dare [11]. It focuses on inter-component communication through intents, also between different apps. It deals with intent filters statically specified in the manifest but also with dynamically registered broadcast receivers. The analyzer does not always detect such receivers in a sound way, because of intrinsic limitations of the tool: no static fields, no reflection, limited string analysis. IccTA [8] identifies sources and sinks with SuSi, like FlowDroid. It translates Dalvik bytecode into Jimple [18], by using Dexpler [3]. It can leverage on both Epicc or its extension IC3 [12] to obtain the inter-component communication methods and their parameters (IC3 is preferred due to its performance). It deals with the dynamic registration of broadcast receivers. Finally, IccTA applies FlowDroid to perform an intra-component taint analysis. The authors report that IccTA performs also an inter-app communication, but do not show any result. In any case, IccTA inherits the same limitations of Epicc and IC3.

```
Intent download = new Intent(this, DownloadService.class);
download.setData(Uri.parse(fileUrl));
startService(download);
```

Fig. 1. Creation of an explicit intent for starting a `DownloadService`.

III. Android Applications and Intents

This section describes the structure of Android apps and their communication model through intents. Examples are from https://developer.android.com/guide/components/intents-filters.html, where the reader can find further detail.

Android supports a modular definition of apps in terms of cooperating *components*, that can be independently developed and replaced as long as their interface is honored. This interface is published in a *manifest*. Components can be user interface screens (*activities*), background tasks (*services*), data resolvers (*content providers*) or generic event listeners (*broadcast receivers*). An app can invoke its own components, but also *exported* components of other apps.

A component starts another component by instantiating an `Intent` object that describes the required task. The intent can be so precise to explicitly name the target component (*explicit* intent) or it can just provide a high-level description, that will go through dynamic resolution of the right component (*implicit* intent). The component is resolved and started by passing the intent to methods `startActivity()` or `startActivityForResult()` (for activities), `startService()` (for services) or `startBroadcast()` (for broadcast receivers). Content providers are started by passing intents to a content resolver.

Intents can carry the arguments qualifying the required task, called *action*. Hence, they are a sort of dynamic component invocation with parameter passing. It follows that any sound static analysis must be aware of how intent resolution works, in order to follow data propagation between components. But this is a complex and highly-dynamic process.

To specify an intent, the programmer sets four properties: (i) an optional, fully qualified name of the target component class. If provided, this string uniquely identifies the target component (explicit intent). Otherwise (implicit intent) component resolution uses the subsequent three properties; (ii) a string specifying the desired action (such as *view*, *send*, *pick*); (iii) the URI referencing data to process, with its MIME type; and (iv) the component kind (category) that will perform the action. Moreover, an intent can carry *extras*, that is, a map from string keys to values, expressing the parameters of the required action. They are not used for component resolution but for the intent payload that must be tracked during a flow analysis.

Fig. 1 shows an example of explicit intent, whose resolution is direct, since it explicitly names the target component (a background service). Fig. 2, instead, shows the creation of an implicit intent to ask the OS to resolve and start a component able to send a text message. If there are more such components, the user will be asked to choose one. Fig. 3 shows a portion of the manifest of an app, whose component `ShareActivity` is willing to perform a message send action, as specified with a

```
1 public class MainActivity extends Activity {
2   public void sendMessage(View view) {
3     String textMessage = telephonyManager.getDeviceId();
4     Intent send = new Intent();
5     send.setAction(Intent.ACTION_SEND);
6     send.putExtra(Intent.EXTRA_TEXT, textMessage);
7     startActivity(send);
8   }
9 }
```

Fig. 2. An activity with an event handler that creates an implicit intent for sending a message. This is inside an app called App1. The constants `Intent.EXTRA_TEXT="android.intent.extra.TEXT"` and `Intent.ACTION_SEND="android.intent.action.SEND"` are hardcoded in the Android OS.

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
  </intent-filter>
</activity>
```

```
1 class ShareActivity extends Activity {
2   protected void onCreate(Bundle savedInstanceState) {
3     Intent intent = getIntent();
4     String data = intent.getString(Intent.EXTRA_TEXT);
5     Log.i(data);
6   }
7 }
```

Fig. 3. A manifest (above) specifying that the activity `ShareActivity` (below) is willing to fulfill `Intent.ACTION_SEND` actions (the value of that constant is explicitly reported in the intent filter). These snippets are from an app called App2.

static intent filter. The started activity recovers (a clone of) the starting intent through its `getIntent()` method, as shown in the same figure. Together, Fig. 2 and 3 form a running example that will be used throughout this article to show a data flow between apps.

Example 1 (Running Example): App1 (Fig. 2) creates an implicit intent with constant action `Intent.ACTION_SEND` and with extra, for the constant key `Intent.EXTRA_TEXT`, bound to sensitive data (the device identifier). It uses this intent to start an activity. App2 registers a matching activity in its manifest (Fig. 3), through an intent filter that dispatches the intent to the component `ShareActivity`. The latter reads the extra for key `Intent.EXTRA_TEXT` and logs it. If both apps are installed in the same device, they could collaborate so that App2 catches the intent sent by App1 and leaks the device identifier to the logs, even though App2 has no explicit access to the device identifier.

IV. Applying Julia's Flow Analysis to Android

Julia [17] is a commercial static analyzer for Java bytecode, based on abstract interpretation [6]. Julia analyzes Java source code, compiled into Java bytecode inside Android Studio. It also analyzes Dalvik bytecode packages (apk), once translated into Java bytecode by using dex2jar[1] (for the code) and apktool (for the manifest). Java bytecode is preprocessed with BCEL[2], getting an object-oriented representation of .class files. Julia has a fixpoint engine that implements denotational and constraint-based analyses. Each framework (*e.g.*, Android 5.0) has information about the runtime. Each checker analyzes a specific property (such as information flow). At the end, Julia issues warnings that can be inspected in the IDE.

[1] https://sourceforge.net/projects/dex2jar

[2] https://commons.apache.org/proper/commons-bcel

A. Flow Analysis for Java

Julia starts the analysis from a set of entry points and builds a semantical model of the execution of a Java program. Namely, all methods reachable, recursively, from the entry points get analyzed. The selection of the entry points can be done in three ways: (i) the entry points are the `main` methods; (ii) the entry points are the public methods (default); (iii) the entry points are the public and the protected methods (library mode).

Among its checkers, Julia includes the injection checker that implements the sound information flow analysis from [7]. It propagates tainted data along all possible information flows. Boolean variables stand for program variables. Boolean formulas model explicit information flows. Namely, their models form a sound overapproximation of all taintedness behaviors for the variables in scope at a given program point. For instance, the abstraction of the `load k` bytecode, that pushes on the operand stack the value of local variable k, is the Boolean formula $(\check{l}_k \leftrightarrow \hat{s}_{top}) \wedge U$, stating that the taintedness of the topmost stack element after this instruction is equal to the taintedness of local variable k before the instruction; all other local variables and stack elements do not change (expressed by a formula U); taintedness before and after an instruction is distinguished by using distinct hats for the variables. There are such formulas for each bytecode instruction. Instructions that might have side-effects (field updates, array writes and method calls) need some approximation of the heap, to model the possible effects of the updates. The analysis of sequential instructions is merged through a sequential composition of formulas. Loops and recursion are saturated by fixpoint. The resulting analysis is a denotational, bottom-up taint analysis implemented through efficient binary decision diagrams [4].

Native methods are a problem for any static analyzer and Julia is no exception. Julia has specific knowledge about the behavior of a few frequently used native methods, such as `System.arraycopy()`. For the others, it applies a worst case assumption. For the injection checker, the latter states that a native method does not affect the heap and can return a tainted value only if at least one of its arguments is tainted. A notable case of native methods are those used for implementing reflection, which allows programmatic code inspection and highly dynamic method calls. Its use, in general, jeopardizes soundness. Julia identifies specific cases where the arguments of reflective calls are constant and statically translates them into method calls. However, the problem remains still open.

Julia uses a dictionary specifying a set of *sources* (for instance, servlet inputs and input methods) and a set of *sinks* (such as SQL query methods, command execution routines, session manipulation methods), so that flows from sources to sinks can be established. The analysis of a source forces the corresponding Boolean variable to be true. At each sink, the analyzer checks if the corresponding Boolean local variable is definitely false. If that is the case, no flow of tainted data into that sink is possible; otherwise, it issues a warning, reporting a potential flow of tainted data into the sink. Hence, Julia's information flow analysis for a program can be expressed as $\textit{taintAnalysis}(\textit{sources}, \textit{sinks}) = \textit{leaks}$, where $\textit{leaks}$ are the program points where Julia issues a leak warning.

B. Application to Android

The selection of the entry points is different in Android, where it is the OS duty to call the event handlers of the components. Hence, Julia scans the Android manifest, looking for XML elements declaring services, activities, receivers and content providers. For each of them, Julia creates a synthetic method that simulates the component's lifecycle (*e.g.*, an activity starts with a call to `onCreate()`, followed by calls to `onStart()`, `onStop()` and `onResume()`). That synthetic method is an entry point for the analysis [14].

A sound flow analysis must match intent creations with serving components, exploiting knowledge about the set of apps installed on the device, and must track the propagation of tainted extras. Both tasks need some analysis of the strings used as intent target component or as keys for extras, that are not necessarily constants. This is complex since there are many alternative ways of creating intents and since the component resolution algorithm is rather involved.

V. The SDLI Tool for Flow Analysis of Android

SDLI, for each app, builds a summary XML file about the flows of intents into and out of the app. By comparing the XML summaries for two apps App1 and App2, it is then possible to determine if tainted data flows out of App1 and enters App2. In that case, SDLI issues a warning.

Example 2: SDLI generates the following summary XML file for App1 in Fig. 2. It states that App1 sends tainted data through an intent created at line 4 of method `sendMessage()` of class `MainActivity`, through the constructor of class `Intent` with no arguments. The intent has action `android.intent.action.SEND` and holds tainted data as extra for key `android.intent.extra.TEXT`:

```
<androidapplication name="App1">
  <intent classname="MainActivity" methodname="sendMessage"
     line=4 origin="new Intent()">
    <outwards>
      <action>android.intent.action.SEND</action>
      <key>android.intent.extra.TEXT</key>
    </outwards>
  </intent>
</androidapplication>
```

Example 3: SDLI generates the following summary XML file for App2 in Fig. 3. It states that App2 extracts an extra for key `android.intent.extra.TEXT` from an intent received from outside the app at line 3 of method `onCreate()` of class `ShareActivity`, by calling method `getIntent()`. The received intent has action `android.intent.action.SEND`:

```
<androidapplication name="App2">
  <intent classname="ShareActivity" methodname="onCreate"
     line=3 origin="getIntent()">
    <inwards>
      <action>android.intent.action.SEND</action>
      <key>android.intent.extra.TEXT</key>
    </inwards>
  </intent>
</androidapplication>
```

```
1: function DUMP_SUMMARY_FOR(app)
2:     // collect a description of the extra's keys that are tainted
3:     tKeys ← ∅
4:     tGetExtra ← ∅
5:     repeat
6:         leaks ← taintAnalysis(sources ∪ tGetExtra, putExtra)
7:         oldtKeys ← tKeys
8:         tKeys ← tKeys ∪
9:             {⟨a, k⟩ | pe ∈ leaks, a ∈ action(pe), k ∈ key(pe)}
10:        tGetExtra ← tGetExtra ∪ tGetExtraFor(tKeys, app)
11:    until oldtKeys = tKeys
12:    leaks ← taintAnalysis(sources ∪ tGetExtra, sinks)
13:    WARN_AT(leaks)
14:    DUMP_OUTWARDS_FLOWS(tKeys, app)
15:    DUMP_INWARDS_FLOWS(app)
16: end function
```

Fig. 4. SDLI's algorithm that analyzes a single *app* and generates its intent summary XML file.

By matching this XML file with that in Ex. 2, SDLI concludes that the outwards tainted intent of App1 matches the inwards intent of App2. Hence, there is a potential flow of tainted data from App1 into App2, through intents.

Ex. 2 and 3 show simple examples of summary files, where a given app only sends an intent or only receives an intent. In general, an app's summary file can contain both and more `<inwards>` and `<outwards>` elements. Moreover, in a few cases, the exact determination of the strings used as `<action>` and `<key>` is not possible, because it is dynamically computed. In those situations, these elements are missing from the XML file and intent matching assumes that an arbitrary string is used instead.

SDLI computes such summary XML files automatically. To determine the tainted outwards intents, it uses an instantiation of Julia's flow analysis where $sources$ are the Android API methods that retrieve sensitive data about the device or its user (such as `TelephonyManager.getDeviceId()`). Similarly, SDLI uses as $sinks$ the Android API methods that write data into files or logs, execute SQL queries, send sms or transmit data through the Internet. Moreover, it uses as sinks also the set $putExtra$ of all program points where data is stored into an intent extra. While the set of sinks is immutable, the set of sources enlarges during the analysis based on how intents are tainted during the execution of an app, in order to track intra-app flows of intents.

Example 4: Consider App1 and App2 from Ex. 1. App1 reads the device identifier by calling the source method `TelephonyManager.getDeviceId()` and App2 leaks information by calling the sink method `Log.i()`. When the analysis of App1 starts, the set of sources is composed only by method `TelephonyManager.getDeviceId()`. The set $putExtra$ includes the call to `putExtra()` in Fig. 2.

Fig. 4 sketches SDLI's algorithm for computing a summary XML file about the intents flowing into and out of an *app*. SDLI runs the flow analysis from Sec. IV on *app*, using the Android *sources* and using the set *putExtra* as sinks (line 6). In this way, it computes (line 9) an abstract description of the extra's keys that might be tainted: pairs of strings for intent action and key. These are determined by functions *action()* and *key()*, that exploit standard allocation-based pointer analysis [1] and string analysis to determine the possible strings used there. When this is not possible, $\top$ is used instead, for soundness. For real world apps, this overapproximation is usually prohibitive in terms of number of false alarms. For this reason, on real world apps the analysis considers only constant strings here. At line 10, SDLI uses the abstract description of the tainted keys to compute which API method calls in *app*, that read an intent extra, could return tainted data. This is the goal of function *tGetExtraFor()*, that uses the manifest of the *app* to determine if the action of the abstract *key* is compatible with the intent filters for the components of the *app*. Also in this case, pointer and string analyses are used to overapproximate the possible keys used for reading the extra. This set *tGetExtra* is then used as sources in the flow analysis at the next iteration (line 6), until no more keys can be considered as tainted (line 11).

Once the fixpoint is reached, SDLI performs a final flow analysis using the Android *sinks* (line 12) and uses its results to generate warnings about intra-app leaks. Then, it dumps the tainted keys as `<outwards>` elements of the XML summary file (line 14). The manifest of the app is used instead for dumping the `<inwards>` elements of the same file (line 15). In both cases, the allocation-based pointer analysis [1] determines the origin of the intent. SDLI generates a summary file for each app and already includes the summaries for the most used Android apps. By comparing pairs of summary files for matching `<outwards>` and `<inwards>` elements, SDLI generates warnings about inter-apps leaks.

VI. Experiments

This section reports experiments with SDLI applied to a set of large real marketplace apps. Hence SDLI analyzes the result of dex2jar and apktool on the .apk packages from the Google Play marketplace. Analyses have been run on an ASUS VivoBook Pro N552VW with an Intel Core i7-6700HQ CPU running at 2.60 GHz, with 16GB RAM and Windows 10 Home Edition at 64 bit.

We applied SDLI to the 50 most popular apps of the Italian Google Play marketplace. These apps, coming from the marketplace, are often heavily obfuscated. Therefore, it was impossible to manually discriminate true and false alarms. For this reason, we also analyzed Telegram, the most popular open source app, whose source can be accessed from its Git repository[3]. Fig. 8 reports the results[4]. **LOC** is the number of non-blank, non-comment lines of source code, as estimated by Julia; **Time** is the full analysis time, including parsing of the code, construction of the call graph and computation of the supporting aliasing and string analyses; **Warns** is the number of warnings. Three analyses went into out of memory.

We checked, manually, a few warnings that SDLI issues on these apps. We used Jadx[5] to decompile the apps into source code.

[3]https://github.com/DrKLO/Telegram.git

[4]We removed three apps because both apktool and dex2jar crashed on them and we could not analyze them.

[5]https://github.com/skylot/jadx

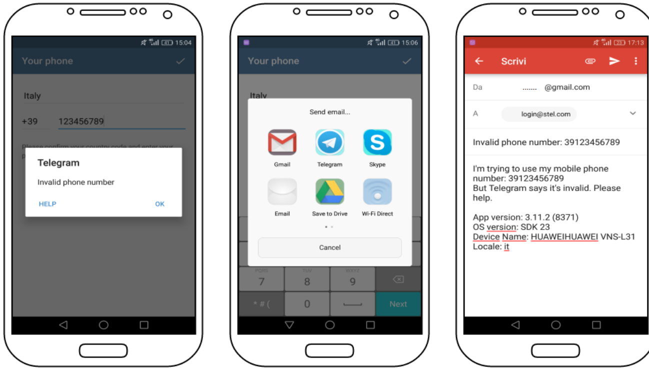

Fig. 5. Invalid number. Fig. 6. App selection. Fig. 7. Gmail is launched.

To determine potential inter-app leakages of sensitive information, we matched the XML summaries for the 50 top apps against that for Intent Analyser[6], a popular Android app developed to let users understand which information can be leaked by intercepting other apps' intents.

The analysis of Telegram yields 188 intra-app alarms. We only report those where we have manually verified that taint information flows into a *putExtra* over an intent. Moreover, we have only considered alarms related to implicit flows, that are more likely to induce security issues. We report an alarm related to explicit intents, as an example.

Telegram v.3.10.1

Telegram[7] is a popular open source messaging app. When it first runs, it asks the user to specify her telephone number. If Telegram deems the number illegal, it complains (Fig. 5) and suggests to contact its technical support. This is achieved by creating an implicit intent with action `Intent.ACTION_SEND`, holding confidential data about the device:

```
Intent mailer = new Intent(Intent.ACTION_SEND);
mailer.putExtra(Intent.EXTRA_TEXT,
  "App version: " + version +
  "\nOS version: SDK " + Build.VERSION.SDK_INT +
  "\nDevice Name: " + Build.MANUFACTURER + Build.MODEL +
  "\nLocale: " + Locale.getDefault());
getParentActivity().startActivity
  (Intent.createChooser(mailer, "Send email..."));
```

The call to `Intent.createChooser()` lets the user choose an app that can serve intents with action `Intent.ACTION_SEND`, if there are more (Fig. 6). The interceptor can catch this intent and leaks the extra. Otherwise, the intent is dispatched to another app that can serve it. Gmail in one such app and can consequently serve the intent. We ran Telegram as in Fig. 7 and actually reproduced the leak. In the next code snippet, an implicit intent gets tainted for key `"output"`. Julia detects that the return value of `generatePicturePath()` is tainted since it can be used to leak the path of the file on the file system. This is, however, the normal way for an app to ask access to the camera:

```
public void openCamera() {
```

[6]https://play.google.com/store/apps/details?id=com.apartapps.android.intentanalyser

[7]https://telegram.org

App	LOC	SDLI Time	SDLI Warns
3	249K	11:29	223
AliBaba	230K	5:53	1
Amazon	354K	27:44	540
Booking	260K	10:28	65
CandyCrash	239K	9:35	129
CatsCrashArena	294K	19:02	265
Chat&Cash	288K	21:07	132
ChickenScream	322K	25:19	268
Clash of Clans	270K	17:35	160
Faceapp	244K	09:07	25
Facebook	231K	OOM	
FacebookLite	266K	13:33	137
FacebookMessenger	233K	8:58	55
FifaMobile	212K	6:18	21
FightList	311K	20:08	207
GooglePhoto	312K	20:40	113
GooglePlayGames	330K	29:07	276
GoogleTranslate	295K	25:37	146
InfoTarga	243K	11:40	139
Instagram	333K	OOM	
Launcher	319K	21:12	259
Musically	195K	5:27	6
MusicDownloader	262K	17:22	170

App	LOC	SDLI Time	SDLI Warns
Netflix	346K	25:32	156
PianoTiles	335K	30:12	492
Pinterest	383K	OOM	
PjMask	254K	13:59	164
Pokemon	248K	11:18	130
Pou	237K	9:32	16
Roll the Ball	323K	26:14	369
Shazam	296K	15:08	37
Slither	290K	22:06	240
Snapchat	141K	2:58	2
Spotify	312K	21:01	51
Subito	232K	8:39	9
Subway Surfers	329K	23:52	325
SuperMario	273K	16:55	229
SuperOptCleaner	222K	7:59	11
Telegram	315K	27:55	188
Tiger Ball	208K	7:20	42
Tim Mobile	253K	11:44	86
Twitter	161K	4:24	0
Vodafone	284K	14:50	73
Waze	265K	9:37	58
WhatsApp	359K	35:43	320
Wind	294K	18:24	115
Wish	332K	20:31	92

Fig. 8. Results of the analysis of the 50 most popular apps on the Italian Google Play marketplace. OOM stands for out of memory.

```
 Intent takePictureIntent = new Intent("IMAGE_CAPTURE");
 File image = AndroidUtilities.generatePicturePath();
 if (image != null) {
  takePictureIntent.putExtra("output", Uri.fromFile(image));
  startActivityForResult(takePictureIntent, ..);
 } ...
}
```

Telegram invokes method `sendLogs()` in case of crash or when an inconsistent runtime state is detected. Such method collects all logs and sends them in an email as attachment. For that purpose, Telegram asks the OS for all apps that can satisfy the action `"android.intent.action.SEND_MULTIPLE"`.

We have matched the XML summary file of Telegram against those of our database of apps and found that Intent Analyser can serve intents with action `"android.intent.action.SEND_MULTIPLE"`. If chosen, it will show the sensitive information held in the intent. This leak can be prevented by avoiding the use of implicit intents. For instance, one can upload the logs to a remote server and share the latter's address by email.

Gmail v.6.11.27.141872707

Gmail[8] specifies, in its manifest, that it is available to serve intents with action `Intent.ACTION_SEND` or `Intent.ACTION_SEND_MULTIPLE`, that get dispatched to its `ComposeActivityGmailExternal` component. That component is obfuscated. Its class extends `ComposeActivityGmail`, that extends `cja`, whose method `a(Message)` is reachable from an event handler of `ComposeActivityGmailExternal`. It accesses the extra for key `Intent.EXTRA_TEXT` and it fills the body of the email by calling method `cja.a(String, boolean, boolean)`:

```
  Intent intent = getIntent();
  String stringExtra2 =
   intent.getCharSequenceExtra("android.intent.extra.TEXT");
  stringExtra2 = Html.toHtml
```

[8]https://play.google.com/store/apps/details?id=com.google.android.gm

```
  (new SpannableString(stringExtra2));
  // fill the textfield of the message of the email
  a(stringExtra2, true, false); ...
```

Being obfuscated, we could not check the method where `stringExtra2` is injected into the email body. However, SDLI signals that the information is leaked and we reproduced the leak in our device. This paves the way to information leaks if other apps send confidential information through intents with action `Intent.ACTION_SEND`, as shown below.

Google Play Games Accounts

Google Play Games uses implicit intents to share data between its components. For instance, method `ClientUiProxyActivity.launchProxyIntent()` broadcasts an intent with action `CLIENT_PROXY`, containing confidential data about the user's accounts, stored there by method `VideoCapturedPopup.handleClick()`:

```
Context localContext = getContext();
Account acc =
 this.mGamesContext.mClientContext.mResolvedAccount; ...
intent.putExtra("com.google.android.gms.games.ACCOUNT", acc);
ClientUiProxyActivity.launchProxyIntent(localContext, intent);
```

That intent was meant to stay inside the app's boundaries. However, being implicit, it might be intercepted by other apps, as it can be verified with Intent Analyser. A malware might consequently gain access to all user accounts.

Discussion

The issues related to implicit intents with action `Intent.ACTION_SEND` do not seem critical, since almost all devices have Gmail installed. Hence the user should explicitly select an untrusted app (malware) with the chooser, instead of Gmail, which seems unlikely. Similarly, the issue related to implicit intents with action `Intent.ACTION_INSERT` does not seem critical: there is, normally, a trusted receiver app for that intent, namely, the app dealing with the phone contacts. A malware might be willing to serve the intent, in which case a chooser will pop up. But it is unlikely that the user will choose the malware, although it might simulate the expected behavior, while actually leaking the information. The issue related to implicit intents with action `CLIENT_PROXY` seems definitely critical, instead. That action is uncommon, it is strictly related to the internal logic of Google Games (rather than to a user request) and there exists just one available receiver among the 50 most popular Android apps: Google Games itself. Hence the OS will automatically redirect the intent to it, without popping up any chooser. But another app (such as Intent Analyser or a malware), might easily intercept the intent. The user will then be faced to a puzzling question about the right app to choose for a non-understandable, never explicitly required action, and might randomly choose the malware. In this case, the programmers of Google Games should have used explicit intents instead.

VII. Conclusion

The new static analysis described in this article tracks information flows through Android intents. Its implementation SDLI, instantiated from Julia's information flow analysis, detects inter-app communications that leak sensitive data. Experiments show that it scales to real world apps and, despite these being heavily obfuscated, allows one to detect actually reproducible leaks of confidential information in some of the most popular apps in the Google Play marketplace.

SDLI confirmed that implicit intents are extremely dangerous, since the receiver is not statically known. Google classified this as a serious security issue and this is why Android 5.0 (API level 21) and later throw an exception at `bindService()` calls with an implicit intent. From the user perspective, an implicit intent carrying confidential extras is a security issue if and only if another app is installed, that catches and leaks it.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] S. Arzt, S. Rasthofer, C. Fritz, 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 *PLDI*, 2014.

[3] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *SOAP*, 2012.

[4] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.

[5] A. Cortesi, P. Ferrara, R. Halder, and M. Zanioli. Combining symbolic and numerical domains for information leakage analysis. *Trans. Computational Science*, pages 98–135, 2018.

[6] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.

[7] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *LPAR*, 2015.

[8] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[9] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In Defense of Soundiness: A Manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[10] A. Mandal, A. Cortesi, P. Ferrara, F. Panarotto, and F. Spoto. Vulnerability analysis of android auto infotainment apps. In *ACM CF*, 2018.

[11] D. Octeau, S. Jha, and P. D. McDaniel. Retargeting Android Applications to Java Bytecode. In *FSE*, 2012.

[12] D. Octeau, D. Luchaup, S. Jha, and P. D. McDaniel. Composite Constant Propagation and its Application to Android Program Analysis. *IEEE Transactions on Software Engineering*, 42(11):999–1014, 2016.

[13] D. Octeau, P. D. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-Component Communication Mapping in Android: An Essential Step towards Holistic Security Analysis. In *USENIX Security*, 2013.

[14] É. Payet and F. Spoto. Static Analysis of Android Programs. *Information & Software Technology*, 54(11):1192–1201, 2012.

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

[16] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2017.

[17] F. Spoto. The Julia Static Analyzer for Java. In *SAS*, 2016.

[18] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode using the Soot Framework: Is It Feasible? In *CC*, 2000.