Using DidFail to Analyze Flow of Sensitive Information in Sets of Android Apps

Will Klieber*, Lori Flynn*, Amar Bhosale, Limin Jia, and Lujo Bauer

*presenting

June 2015
Overview

Problem: Sensitive/private information can be leaked by apps on smartphones.
- Precise detection on Android is made difficult by communication between components of apps.
- Malicious apps could evade detection by collusion or by exploiting a leaky app using *intents* (messages to Android app components) to pass sensitive data.

Goal: Precisely detect undesired flows across multiple Android components.
- Remedies if such flows are discovered:
  - At present: Refuse to install app
  - Future work: Block undesired flows

Our Tool (*DidFail*):
- Input: set of Android apps (APK files)
- Output: list of flows of sensitive information

Major Achievements:
- First published static taint flow analysis for app sets (not just single apps)
- Fast user response: two-phase method uses phase-1 precomputation
Introduction

One billion Android devices (phones and tablets) estimated sold in 2014.\(^1\)

Goal: Detect malicious apps that leak sensitive data.

- E.g., leak contacts list to marketing company.
- “All or nothing” permission model.

Apps can collude to leak data.

- Evades precise detection if only analyzed individually.

\(^1\) Gartner Report: [http://www.gartner.com/newsroom/id/2665715](http://www.gartner.com/newsroom/id/2665715)
Introduction: Android

Android apps have four types of components:

- Activities
- Services
- Content providers
- Broadcast receivers

**Intents** are messages to components.

- Explicit or implicit designation of recipient

Components declare **intent filters** to receive implicit intents.

Matched based on properties of intents, e.g.:

- Action string (e.g., “android.intent.action.VIEW”)
- Data MIME type (e.g., “image/png”)
Introduction

Taint Analysis tracks the flow of sensitive data.

- Can be static or dynamic.
  - Static analysis: Analyze the code without running it.
  - Dynamic analysis: Analyze the program by running it.
- Our analysis is static.

Our analysis is built upon existing Android static analyses:

- FlowDroid [1]: finds intra-component information flow
- Epicc [2]: identifies intent specifications


Our Contribution

We developed the **DidFail** static analyzer ("Droid Intent Data Flow Analysis for Information Leakage").

- Finds flows of sensitive data across app boundaries.
- Source code available at: (or google “DidFail CERT”) http://www.cert.org/secure-coding/tools/didfail.cfm

Two-phase analysis:

1. Analyze each app in isolation.
2. Use the result of Phase-1 analysis to determine inter-app flows.

We tested our analyzer on sets of apps.
**Terminology**

**Definition.** A *source* is an external resource (external to the component/app, not necessarily external to the phone) from which data is read.

**Definition.** A *sink* is an external resource to which data is written.

For example,
- **Sources**: Device ID, contacts, photos, location (GPS), intents, etc.
- **Sinks**: Internet, outbound text messages, file system, intents, etc.

**Definition.** Data is *tainted* if it originated from a (sensitive) source.
Analysis of Android App Sets: Sensitive Dataflow

- If an undesired flow is discovered:
  - User might refuse to install app
  - App store might remove app

Previous tools: taint flow in single component
- Intents can be treated as sources/sinks.
- But cannot precisely identify full flows involving multiple components.

Malicious developer strategy:
- Hide from tools by using multiple apps for tainted data flow (launder)
- Colluding apps, or combination leaky app and malicious app

DidFail:
- Defeat multiple-app strategy, detect full tainted flows
- First published static taint flow analysis for app sets
- Fast user response: 2 phases
Motivating Example

App *SendSMS.apk* sends an **intent** (a message) to *Echoer.apk*, which sends a **result** back.

- *SendSMS.apk* tries to launder the taint through *Echoer.apk*.
- Pre-existing static analysis tools could not precisely detect such inter-app data flows.
Analysis Design

**Phase 1**: Each app analyzed once, in isolation.
- **FlowDroid**: Finds tainted dataflow from sources to sinks.
  - Received intents are considered sources.
  - Sent intent are considered sinks.
- **Epicc**: Determines properties of intents.
- Each intent-sending call site is labelled with a unique *intent ID*.

**Phase 2**: Analyze a set of apps:
- For each intent sent by a component, determine which components can receive the intent.
- Generate & solve taint flow equations.
Running Example

Three components: $C_1$, $C_2$, $C_3$.
$C_1 = \text{SendSMS}$
$C_2 = \text{Echoer}$
$C_3$ is similar to $C_1$

For $i \in \{1, 3\}$:
- $C_i$ sends data from $src_i$ to component $C_2$ via intent $I_i$.
- $C_2$ reads data from intent $I_i$ and echoes it back to $C_i$.
- $C_i$ reads data from the result and writes it to $sink_i$.

- $sink_1$ is tainted with only $src_1$.
- $sink_3$ is tainted with only $src_3$. 
Running Example

Notation:

- “src \xrightarrow{C} sink”: Flow from src to sink in C.
- “I(C_{TX}, C_{RX}, id)”: Intent from $C_{TX}$ to $C_{RX}$ with ID id.
- “R(I)”: Response (result) for intent I.
- “T(s)”: Set of sources with which s is tainted.
Running Example

Notation:

- “src $\rightarrow$ sink”: Flow from src to sink in C.
- “$I(C_{TX}, C_{RX}, id)$”: Intent from $C_{TX}$ to $C_{RX}$ with ID id.
- “$R(I)$”: Response (result) for intent I.
- “$T(s)$”: Set of sources with which s is tainted.

Final Sink Taints:

- $T(sink_1) = \{src_1\}
- $T(sink_3) = \{src_3\}$
Phase-1 Flow Equations

Analyze each component separately.

Phase 1 Flow Equations:

\[
\begin{align*}
src_1 & \xrightarrow{C_1} I(C_1, *, id_1) \\
R(I(C_1, *, *)) & \xrightarrow{C_1} sink_1 \\
I(\ast, C_2, \ast) & \xrightarrow{C_2} R(I(\ast, C_2, \ast)) \\
src_3 & \xrightarrow{C_3} I(C_3, *, id_3) \\
R(I(C_3, *, *)) & \xrightarrow{C_3} sink_3
\end{align*}
\]

Notation

- “\(src \xrightarrow{C} sink\)”: Flow from \(src\) to \(sink\) in \(C\).
- “\(I(C_{TX}, C_{RX}, id)\)”: Intent from \(C_{TX}\) to \(C_{RX}\) with ID \(id\).
- “\(R(I)\)”: Response (result) for intent \(I\).
- An asterisk (“\(\ast\)”) indicates an unknown component.
Phase-2 Flow Equations

Instantiate Phase-1 equations for all possible sender/receiver pairs.

### Phase 1 Flow Equations:

<table>
<thead>
<tr>
<th>Source</th>
<th>Phase</th>
<th>Flow Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>src₁</td>
<td>C₁</td>
<td>I(C₁, *, id₁)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(I(C₁, *, *)) → C₁ → sink₁</td>
</tr>
<tr>
<td></td>
<td>C₂</td>
<td>I(*, C₂, <em>) → R(I(</em>, C₂, *))</td>
</tr>
<tr>
<td></td>
<td>C₃</td>
<td>I(C₃, *, id₃)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(I(C₃, *, *)) → C₃ → sink₃</td>
</tr>
</tbody>
</table>

### Phase 2 Flow Equations:

<table>
<thead>
<tr>
<th>Source</th>
<th>Phase</th>
<th>Flow Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>src₁</td>
<td>C₁</td>
<td>I(C₁, C₂, id₁)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(I(C₁, C₂, id₁)) → C₁ → sink₁</td>
</tr>
<tr>
<td></td>
<td>C₂</td>
<td>I(C₁, C₂, id₁) → R(I(C₁, C₂, id₁))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I(C₃, C₂, id₃) → C₂ → R(I(C₃, C₂, id₃))</td>
</tr>
<tr>
<td></td>
<td>C₃</td>
<td>I(C₃, C₂, id₃)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(I(C₃, C₂, id₃)) → C₃ → sink₃</td>
</tr>
</tbody>
</table>

**Notation**

- "src → sink": Flow from src to sink in C.
- "I(C_TX, C_RX, id)": Intent from C_TX to C_RX with ID id.
- "R(I)": Response (result) for intent I.

**Manifest and Epicc info (not shown) are used to match intent senders and recipients.**
Phase-2 Taint Equations

For each flow equation $src \rightarrow sink$, generate taint equation $T(src) \subseteq T(sink)$.

**Phase 2 Flow Equations:**

- $src_1 \xrightarrow{C_1} I(C_1, C_2, id_1)$
- $R(I(C_1, C_2, id_1)) \xrightarrow{C_1} sink_1$
- $I(C_1, C_2, id_1) \xrightarrow{C_2} R(I(C_1, C_2, id_1))$
- $I(C_3, C_2, id_3) \xrightarrow{C_2} R(I(C_3, C_2, id_3))$
- $src_3 \xrightarrow{C_3} I(C_3, C_2, id_3)$
- $R(I(C_3, C_2, id_3)) \xrightarrow{C_3} sink_3$

**Phase 2 Taint Equations:**

- $T(src_1) \subseteq T(I(C_1, C_2, id_1))$
- $T(R(I(C_1, C_2, id_1))) \subseteq T(sink_1)$
- $T(I(C_1, C_2, id_1)) \subseteq T(R(I(C_1, C_2, id_1)))$
- $T(I(C_3, C_2, id_3)) \subseteq T(R(I(C_3, C_2, id_3)))$
- $T(src_3) \subseteq T(I(C_3, C_2, id_3))$
- $T(R(I(C_3, C_2, id_3))) \subseteq T(sink_3)$

**Notation**

- “$src \xrightarrow{C} sink$”: Flow from $src$ to $sink$ in $C$.
- “$I(C_{TX}, C_{RX}, id)$”: Intent from $C_{TX}$ to $C_{RX}$ with ID $id$.
- “$R(I)$”: Response (result) for intent $I$.
- “$T(s)$”: Set of sources with which $s$ is tainted.

**Then, solve.**

If $s$ is a non-intent source, then $T(s) = \{s\}$.
Phase 1

Original APK → TransformAPK → Extract manifest → Epicc → FlowDroid (modified)

App 1 → Phase 1 → App 2 → Phase 1 → App 3 → Phase 1 → App n → Phase 1

Phase 2
Implementation: Phase 1

APK Transformer

• Assigns unique Intent ID to each call site of intent-sending methods.
  o Enables matching intents from the output of FlowDroid and Epicc
• Uses Soot to read APK, modify code (in Jimple), and write new APK.

• Problem: Epicc is closed-source. How to make it emit Intent IDs?
• Solution (hack): Add putExtra call with Intent ID.
Implementation: Phase 1

FlowDroid Modifications:

• Extract intent IDs inserted by APK Transformer, and include in output.
• When sink is an intent, identify the sending component.
  o In `base.startActivity`, assume `base` is the sending component.
• For deterministic output: Sort the final list of flows.
Implementation: Phase 2

Phase 2

- Input: Phase 1 output.
- Generate and solve the data-flow equations.
- Output:
  1. Directed graph indicating information flow between sources, intents, intent results, and sinks.
  2. Taintedness of each sink.
Testing DidFail analyzer: App Set 1

SendSMS.apk
• Reads device ID, passes through Echoer, and leaks it via SMS

Echoer.apk
• Echoes the data received via an intent

WriteFile.apk
• Reads physical location (from GPS), passes through Echoer, and writes it to a file

Flows found by DidFail

- `getDeviceId` → `SendSMS` → `startActivityForResult`
- `getIntent` → `Echoer` → `setResult`
- `onActivityResult` → `SendSMS` → `sendTextMessage`
- `getLastKnownLocation` → `WriteFile` → `startActivityForResult`
- `getIntent` → `Echoer` → `setResult`
- `onActivityResult` → `WriteFile` → `write`
Limitations

Unsoundness

• Inherited from FlowDroid/Epicc
  – Native code, reflection, etc.

• Shared static fields
  – Partially addressed by Jonathan Burkert, but with scalability issues

• Implicit flows

• Originally only considered activity intents
  – Students added partial support for services and broadcast receivers.

Imprecision

• Inherited from FlowDroid/Epicc

• Didn’t consider permissions when matching intents

• All intents received by a component are conflated together as a single source
Use of Two-Phase Approach in App Stores

We envision that the two-phase analysis can be used as follows:

- An app store runs the phase-1 analysis for each app it has.
- When the user wants to download a new app, the store runs the phase-2 analysis and indicates new flows.
- Fast response to user.

**Policy guidance/enforcement, for usability.**
Usability: Policies to Determine Allowed Flows

Policy: Prohibit flow from Src₁ to Sink₃

Example 1

Policy:
  Prohibit flow from Src₁ to Sink₃

Example 2

Policies could come from:
• App store
• Security system provider
• Employer
• User option

C₁ C₂ C₃
Src₁ Sink₁ Src₃ Sink₃
I(C₁, C₂, id₁)
I(C₃, C₂, id₂)

Compliant

I(C₁, C₂, id₁)
I(C₃, C₂, id₂)

Noncompliant
DidFail vs IccTA

IccTA was developed (at roughly the same time as DidFail)
IccTA uses a one-phase analysis
  • IccTA is more precise than DidFail’s two-phase analysis.
    - More context-sensitive
    - Less overestimation of taints reaching sinks
  • Two-phase DidFail analysis allows fast 2nd-phase computation.
    - Pre-computed Phase-1 analysis done ahead of time
    - User doesn’t need to wait long for Phase-2 analysis

Typical time for simple apps:
  • DidFail: 2 sec (2nd phase)
  • IccTA: 30 sec

Working together now! Ongoing collaboration between IccTA and DidFail teams
Analysis of Android App Sets: Sensitive Dataflow

Novel Android static dataflow analysis “DidFail” combines precise single-component taint analysis (FlowDroid) and intent analysis (Epicc).

- **Phase 1**: Each app analyzed once, in isolation
  - Examine flow of tainted data from sources to sinks (including intents)
  - Examines intent properties to match senders and receivers
- **Phase 2**: For a particular set of apps
  - Generate taint flow equations
  - Iteratively solve equations
  - Fast!

**Goal: enforce confidentiality and integrity**

Source code: http://www.cert.org/secure-coding/tools/didfail.cfm
Installing DidFail

Main DidFail website
  • http://www.cert.org/secure-coding/tools/didfail.cfm

Detailed install instructions are on the download website
  • https://www.cs.cmu.edu/~wklieber/didfail/install-latest.html

There are 3 branches
  • Static fields (Dec. 2014)
  • Services and broadcast receivers (Dec. 2014)
  • Improved DEX conversion (Nov. 2014)
Running DidFail

To run DidFail (both phases 1 and 2):

```
$ run-didfail.sh OUT_DIR APK1 ... APKn
```

Running just parts of phase 1:

- The scripts for running parts of Phase 1 independently are available in the latest versions of the three branches in the repository.
- First, set up environment variables in your Bash shell:

  ```
  $ source paths.local.sh
  ```

- Running APK Transformer:

  ```
  $ run-transformer.sh OUT_DIR APK
  ```

- Running FlowDroid:

  ```
  $ run-indep-flowdroid.sh OUT_DIR APK
  ```

- Running Epicc:

  ```
  $ run-indep-epicc.sh OUT_DIR APK
  ```

- Extracting manifest file (to stdout):

  ```
  $ extract-manifest.sh APK
  ```

Running Phase 2:

```
$ python taintflows.py phasel_output_files --js jsonfile --gv graphfile
[--quiet]
```
Phase-1 Output from FlowDroid (Echoer Toy App)

3 possible flows to sinks found

3 <flow>
4 <sink method="&lt;android.util.Log: int i(java.lang.String,java.lang.String)&gt;"/>
5 <source method="&lt;android.app.Activity: android.content.Intent getIntent()&gt;" component="org.cert.echoer.MainActivity">
6 <in>getDataFromIntent</in>
7 </source>
8 <source method="&lt;android.os.Bundle: java.lang.String getString(java.lang.String)&gt;" component="org.cert.echoer.MainActivity">
9 <in>getDataFromIntent</in>
10 </source>
11 </flow>

12 <flow>
13 <sink method="&lt;android.util.Log: int i(java.lang.String,java.lang.String)&gt;"/>
14 <source method="&lt;android.app.Activity: android.content.Intent getIntent()&gt;" component="org.cert.echoer.MainActivity">
15 <in>getDataFromIntent</in>
16 </source>
17 </flow>

18 <flow>
19 <sink method="&lt;android.app.Activity: void setResult(int,android.content.Intent)&gt;" is-intent-result="1" component="org.cert.echoer.MainActivity">
20 <in>onClickListener</in>
21 </sink>
22 <source method="&lt;android.app.Activity: android.content.Intent getIntent()&gt;" component="org.cert.echoer.MainActivity">
23 <in>getDataFromIntent</in>
24 </source>
25 </flow>
Phase-1 Output from FlowDroid: One XML `<flow>` for Echoer

```xml
<flow>
  <sink method="&lt;android.util.Log: int i(java.lang.String, java.lang.String)"/>
  <source type="&lt;android.app.Activity: android.content.Intent getIntent()" component="org.cert.echoer.MainActivity">
    <in>getDataFromIntent</in>
  </source>
  <source type="&lt;android.os.Bundle: java.lang.String getString(java.lang.String)" component="org.cert.echoer.MainActivity">
    <in>getDataFromIntent</in>
  </source>
</flow>
```
Phase-1 Output from Epicc (SendSMS Toy App)

Epicc provides precision about fields in intents sent

485 The following ICC values were found:
486 - org/cert/sendsms/Button1Listener/onClick(Landroid/
   487 Intent value: 1 possible value(s):
488 Action: android.intent.action.SEND, Type: text/plain,
   Extras: [newField_6, secret]
GraphViz output for DroidBench app set

Int3 = I(IntentSink2.apk, IntentSource1.apk, id3)
Int4 = I(IntentSource1.apk, IntentSink1.apk, id4)
Res8 = R(Int4)
Src15 = getDeviceId
Snk13 = Log.i

Some flows:

- $\text{Src15} \xrightarrow{\text{IntentSink2}} \text{Int3} \xrightarrow{\text{IntentSource1}} \text{Snk13}$
- $\text{Src15} \xrightarrow{\text{IntentSink2}} \text{Int3} \xrightarrow{\text{IntentSource1}} \text{Int4} \xrightarrow{\text{IntentSink1}} \text{Res8} \xrightarrow{\text{IntentSource1}} \text{Snk13}$
- $\text{Src15} \xrightarrow{\text{IntentSink1}} \text{Res8} \xrightarrow{\text{IntentSource1}} \text{Snk13}$
Phase-2 Output: JSON-format (excerpts)

```
1. {
2.   "Flows": [
4.     "org.cert.sendsms",
6.   ],
7.   [
9.     null,
10.    "Intent(tx=('[org.cert.sendsms]', 'MainActivity'),
11.       rx=('[org.cert.echoer]', 'MainActivity'), intent_id='newField_6')",
12.   ],
13.   [
14.     "Intent(tx=('[org.cert.sendsms]', 'MainActivity'),
15.       rx=('[org.cert.echoer]', 'MainActivity'), intent_id='newField_6')",
16.     null,
18.   ],
```

Phase-2 Output: JSON-format (excerpts)

19. "Taints": {
20.   "Intent(tx=('org.cert.sendsms', 'MainActivity'),

21.     rx=('org.cert.echoer', 'MainActivity'),

22.     intent_id='newField_6')": [

24.     "Sink: <android.telephony.SmsManager:

25.       void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,


28.   },}
For More Information

Secure Coding Initiative
- Will Klieber, Lori Flynn
  {weklieber,lflynn}@cert.org

Web
- www.cert.org/secure-coding
- www.securecoding.cert.org

U.S. Mail
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Subscribe to the CERT Secure Coding eNewsletter
mailto: info@sei.cmu.edu