Software Engineering Institute
Carnegie Mellon University

DidFail: Coverage and Precision Enhancement

Karan Dwivedi
Hongli Yin
Pranav Bagree
Xiaoxiao Tang
Lori Flynn
William Klieber
William Snavely

July 2017

TECHNICAL REPORT
CMU/SEI-2017-TR-007

CERT Division

Table of Contents

List of Figures

Acknowledgments

We thank Professor Patrick Tague of Carnegie Mellon University for allowing the graduate student team from his Mobile Security class (Karan Dwivedi, Hongli Yin, Pranav Bagree, and Xiaoxiao Tang) to work on this project in the Fall semester of 2015.

Abstract

This report describes recent enhancements to Droid Intent Data Flow Analysis for Information Leakage (DidFail), the CERT static taint analyzer for sets of Android apps. The enhancements are new analytical functionality for content providers, file accesses, and dynamic broadcast receivers. Previously, DidFail did not analyze taint flows involving ContentProvider components; however, now it analyzes taint flows involving all four types of Android components. The latest version of DidFail tracks taint flow across file access calls more precisely than it did in prior versions of the software. DidFail was also modified to handle dynamically declared BroadcastReceiver components in a fully automated way, by integrating it with a recent version of FlowDroid and working to fix remaining un-analyzed taint flows. Finally, a new command line argument optionally disables static field analysis in order to reduce DidFail's memory usage and analysis time. These new features make DidFail's taint tracking more precise (for files) and more comprehensive for dynamically registered BroadcastReceiver and ContentProvider components. We implemented the new features and tested them on example apps that we developed and on real-world apps from different categories in the Google Play app store.

1 Introduction

Droid Intent Data Flow Analysis for Information Leakage (DidFail) is a static analyzer that was developed by the CERT Division of the Software Engineering Institute (SEI) at Carnegie Mellon University. DidFail detects possible flows of sensitive information in sets of Android apps [1,8].

This technical report describes recent enhancements to the DidFail analyzer, newly developed test apps, and test results. The enhancements include new analytical functionality for Content Providers, file accesses, and dynamically registered BroadcastReceivers. Additionally, we merged the two software branches of the previous release.

Detection of potential taint flows can be used to protect sensitive data, identify leaky apps, and identify malware. In the terminology for DidFail, a *sink* is an external-to-the-app resource to which data is written; a *source* is an external-to-the-app resource from which data is read. We say a piece of data is *tainted* if it originates from a sensitive source. For example, a sensitive source could be the device ID or GPS location of the device; a sink could be a Short Message Service (SMS) message to be sent. The goal of DidFail is to find any possible path through which data flows from a source to a sink, by doing static data flow analysis on a set of Android apps (Android Package Kits, or APKs).

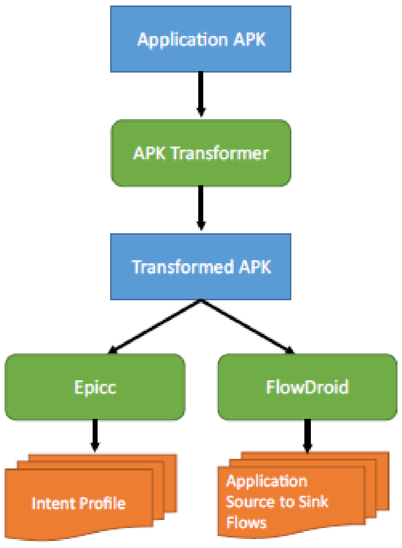


Figure 1.1: Overview of DidFail Phase 1

DidFail works using two phases, the first of which is shown in Figure 1.1.

In **Phase 1**, each APK is fed into the *APK Transformer*, a tool that annotates intent-related function calls with information that uniquely identifies individual cases where intents are used in the application. Once completed, the transformed APK is passed to two other tools: FlowDroid [7] and Epicc [11].

The FlowDroid tool performs static taint tracking in Android applications. Given a set of method signatures that correspond to taint sources and sinks, FlowDroid conservatively propagates taint from sources in the application and reports all flows from sensitive sources to sinks. Sources include function calls that access sensitive information in Android, such as `getLatitude()` and `getSimSerialNumber()`. Sinks include function calls that exfiltrate information, such as `Log.d` and `FileOutputStream.write`. In addition, reads from received intents are treated as sources and writes to intents are treated as sinks. One example of output from FlowDroid might identify that an application reads contact information from a source, then sends it as part of an intent. Another example of output might identify that an application reads information from an intent, then sends it via SMS message.

Epicc [11] performs static analysis to map out inter-component communication within an Android application. While this analysis is mainly used to understand how parts of a single application work together, it can also discover which portions of the application are externally accessible via either explicit or implicit intents. While FlowDroid is useful for understanding flows within an application, Epicc reveals the interfaces that can be used for an application to communicate with other applications.

Phase 1 of DidFail can be performed on one application at a time. Once completed, it does not need to be run again.

In **Phase 2**, DidFail combines the Phase 1 output of multiple applications to determine how specific apps in a set can interact. DidFail analyzes data flows between apps that can occur through intents, files, and static fields, eventually discovering and reporting potential full taint flows. A full taint flow is a data flow from an external source to an external sink that goes through at least one Android app.

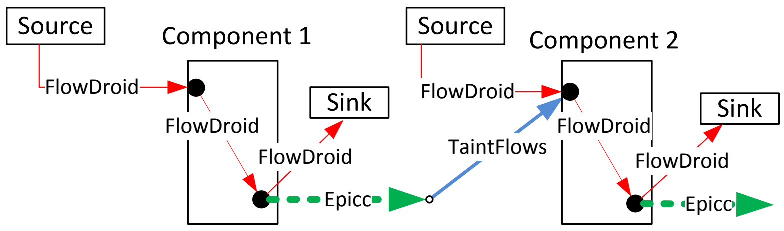


Figure 1.2: Data Flow Taint Tracking

Android has a powerful and complex communication system for sending data between apps. Simpler static analyses do not analyze taint flows across multiple apps. Malicious apps could take advantage of this to avoid detection despite using sensitive information from apps with data leaks. Alternatively, they may *launder* the data (to avoid detection) by sending sensitive information to a colluding app first instead of directly leaking it off the phone.

ICCTA [9], like DidFail, performs analysis of flows of sensitive data in Android app sets. However, ICCTA analyzes a set of apps monolithically in one step, as opposed to the two-phase DidFail analysis.

One goal of the ongoing DidFail work is to eventually provide app stores, security vendors, and security researchers with a practical technique and tool to determine if unwanted taint flows could occur [6]. For practical use, the technique must work quickly and precisely; it must also perform a comprehensive analysis.

Using DidFail's two-phase analysis approach would allow a security vendor or app store to respond quickly (within 1 second or so in typical cases) to user requests to install a new app. Although the first phase of the analysis is the most time-consuming part, it can be pre-computed for all apps in an app store. For instance, an app store could respond by providing the app if no problematic taint flows could happen. It could respond with policy-compliant options if adding the new app "as is" to the current app set would enable a policy-violating taint flow. The DidFail enhancements described in this report are intended to make progress toward this goal, not to complete it.

2 Motivation

Android's permission system is an important feature of its security. For versions of the Android operating system prior to Android Marshmallow, Android restricts users to an all-or-nothing approach to permissions that are used by an application. The Android market is highly fragmented: except for Android phones sold directly by Google (e.g., Nexus and Pixel), there is no assurance from Google that an Android phone will receive any security or other updates to its operating system. As of January 9, 2017, only 30% of currently used Android smartphones use Android Marshmallow OS or later [3] and thus have more control over their per-app permissions. For the other 70% of Android systems, users who wish to install an app must first agree to grant all permissions that it requests. Even for the 30% of users with greater permissions controls, many will simply approve all requested permissions; others may try to be cautious with permissions but still grant some that allow unwanted taint flow. This results in untrusted apps with permissions to access sensitive data or communications channels such as location, device ID, contacts, and network communications access [2].

One assumption made by many Android users is that an app that lacks a permission to access a resource cannot access that resource in any way. In reality, the Android operating system enforces a permission requirement by checking if the app has the permission when the app requests a particular resource. An app with permission to access the resource can access it freely. However, the Android permission model does not enforce permissions when sensitive data is communicated between applications. Thus, sensitive information can be leaked from an Android phone. Apps can collude with other apps and share access that only one of the apps has permission for. Unknown to the Android user, sensitive data could be exfiltrated to the Internet, if an app with permission to communicate with the Internet intercepts data from a leaky app that handles sensitive data.

DidFail addresses this problem by providing a way to detect potential taint flows in app sets. The latest DidFail enhancements are described in the rest of this report.

3 Goals

3.1 File System Taint Tracking

The previous version of DidFail did not report taint flows through the file system, although it did use the file system as a source and a sink (i.e., as endpoints for detected potential taint flows). DidFail uses FlowDroid, which considers the whole file system as a single element in the data flow. FlowDroid does not track which file is written to or read from, resulting in false positives (if all writes are considered to taint all reads) or false negatives (if no writes are considered to taint any reads). The current version of DidFail improves taint flow tracking through the file system to solve this problem.

3.2 Taint Tracking of Content Providers

The previous version of DidFail did not support taint tracking through Content Providers. Thus, if a malicious flow results in unintended data leakage by passing data via one or more Content Providers, DidFail did not detect the flow at all. Content Providers are one of four types of Android components, and are used by apps for storing and retrieving data. The Android operating system has its own Content Providers, which can be used by apps. Furthermore, Content Providers can also be custom, i.e., application developers can create their own Content Providers as part of their application. The previous version of DidFail did taint flow analysis for data flows involving the other three Android components, but not for Content Providers. Content Providers are an integral part of the Android ecosystem and it is important to track taint flows involving them. We added support for taint tracking through Content Providers.

3.3 Support for Dynamically Registered BroadcastReceivers (Dynamic Receivers)

The previous version of DidFail did not provide full support for dynamically registered BroadcastReceivers (also known as "dynamic receivers"). An app can declare the broadcasts it wishes to receive statically in its manifest or dynamically with the `registerReceiver()` method. Dynamically registered receivers do not appear in the manifest, but they do appear in the output of Epicc. An entry starts with the line `Type: Android.content.BroadcastReceiver`. An app can un-register a dynamically registered receiver (and thereby stop it from receiving broadcasts) by calling the `Context.unregisterReceiver()` method.

DidFail does not consider calls to `unregisterReceiver()`; thus, if a receiver cannot be live when a tainted broadcast is sent, DidFail can produce a false alarm. DidFail handles dynamically registered BroadcastReceivers by adding a dummy static declaration of the BroadcastReceiver to the manifest file, so that FlowDroid can analyze it. In previous versions of DidFail, this dummy declaration needed to be manually added after analysis of the code showed a BroadcastReceiver was dynamically registered.

The current version of DidFail automates this process and eliminates the need for manual edits.

3.4 Toggle Computation-Intensive Analysis

Static field support was added to DidFail in 2015 [1]. While this feature makes it possible to identify information leaks involving data flows through static fields, it leads to higher memory usage and longer analysis times. As a result, in previously-performed experimental tests, 66 out of 100 test apps which otherwise could be analyzed by DidFail either timed out or ran out of heap space [1].

The current version of DidFail adds a toggle to optionally skip static field-related analysis, which lowers memory usage and shortens analysis times.

4 Implementation

This section details our implementation of the DidFail enhancements outlined in Section 3.

4.1 Taint Tracking of File System

One of our goals is to enhance the DidFail to track taint flows that flow through the file system. Consider the following two scenarios of taint flow through the file system:

- **Scenario 1** involves two apps, *WriteFile.apk* and *ReadFile.apk*. *WriteFile.apk* reads location information and writes it to a file named `file.txt` on the smartphone's SD card. *ReadFile.apk* reads another file named `file2.txt` and sends it out through SMS.
- **Scenario 2** involves two apps, *WriteFile.apk* and *ReadSameFile.apk*. *WriteFile.apk* reads location information and writes it to a file named `file.txt` on the smartphone's SD card. *ReadSameFile.apk* reads the location data from the same file and sends it out via SMS.

The previous version of DidFail could not distinguish between Scenario 1 and Scenario 2, leading to either false positives (if all writes taint all reads) or false negatives (if writes never taint reads). Our DidFail enhancement solves this problem by distinguishing between these two scenarios.

We modified both of the DidFail phases to make data flow analysis through files more precise. In Phase 1, we first identify the file-system-related APIs that are potential sources and sinks, as shown in Figure 4.1 and Figure 4.2. When DidFail finds a sink or source, we first determine if the sink or source is on the list of APIs related to the file system. If so, we try to extract the file path and add it to the FlowDroid output. We currently handle cases precisely only if the file path is a compile-time constant. We also modified the Phase 2 code to make it compatible with the more precise file-level taint flow tracking. We now chain taint flows in different components by identifying whether any two apps have read or write operations on the same file.

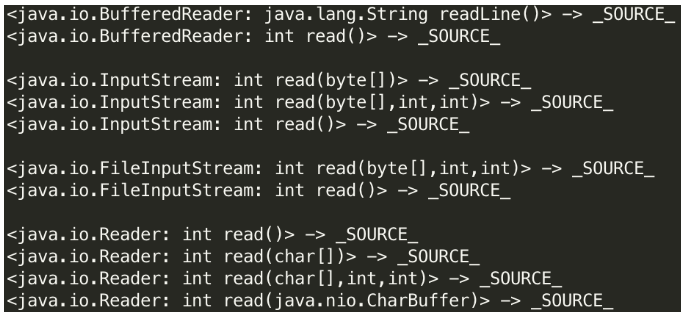

```
<java.io.BufferedReader: java.lang.String readLine()> -> _SOURCE_
<java.io.BufferedReader: int read()> -> _SOURCE_

<java.io.InputStream: int read(byte[])> -> _SOURCE_
<java.io.InputStream: int read(byte[],int,int)> -> _SOURCE_
<java.io.InputStream: int read()> -> _SOURCE_

<java.io.FileInputStream: int read(byte[],int,int)> -> _SOURCE_
<java.io.FileInputStream: int read()> -> _SOURCE_

<java.io.Reader: int read()> -> _SOURCE_
<java.io.Reader: int read(char[])> -> _SOURCE_
<java.io.Reader: int read(char[],int,int)> -> _SOURCE_
<java.io.Reader: int read(java.nio.CharBuffer)> -> _SOURCE_
```

Figure 4.1: Sources Related to File System

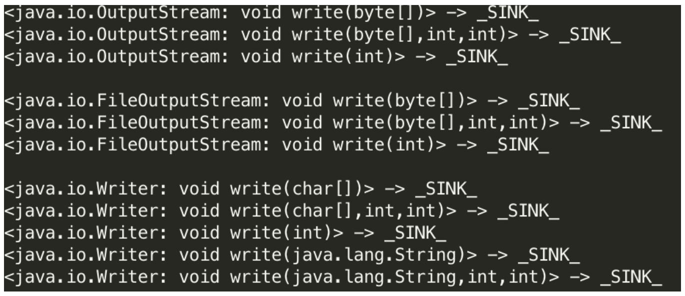
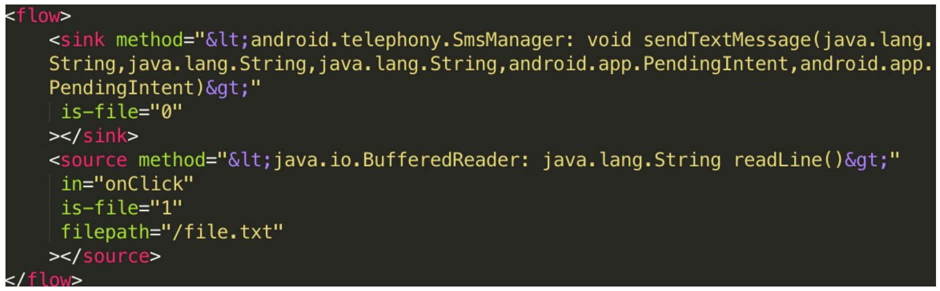
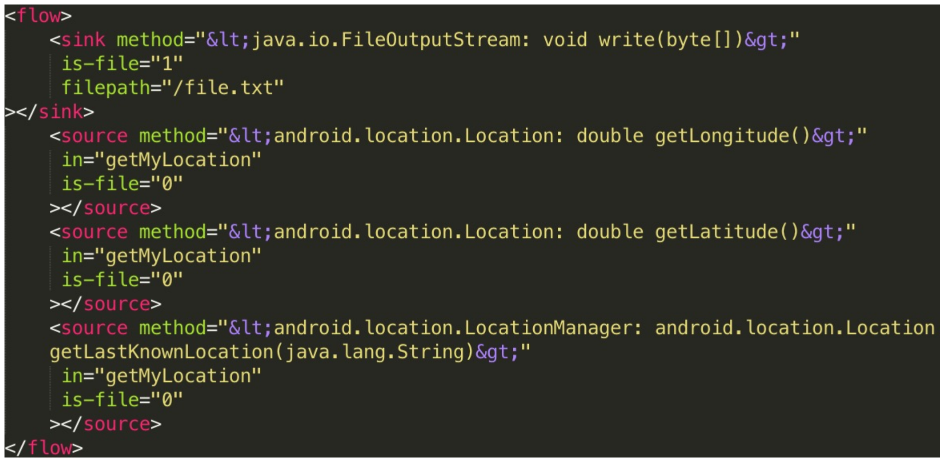

```
<java.io.OutputStream: void write(byte[])> -> _SINK_
<java.io.OutputStream: void write(byte[],int,int)> -> _SINK_
<java.io.OutputStream: void write(int)> -> _SINK_

<java.io.FileOutputStream: void write(byte[])> -> _SINK_
<java.io.FileOutputStream: void write(byte[],int,int)> -> _SINK_
<java.io.FileOutputStream: void write(int)> -> _SINK_

<java.io.Writer: void write(char[])> -> _SINK_
<java.io.Writer: void write(char[],int,int)> -> _SINK_
<java.io.Writer: void write(int)> -> _SINK_
<java.io.Writer: void write(java.lang.String)> -> _SINK_
<java.io.Writer: void write(java.lang.String,int,int)> -> _SINK_
```

Figure 4.2: Sinks Related to File System



```
<flow>
    <sink method="&lt;android.telephony.SmsManager: void sendTextMessage(java.lang.
    String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app.
    PendingIntent)&gt;"
     is-file="0"
    ></sink>
    <source method="&lt;java.io.BufferedReader: java.lang.String readLine()&gt;"
     in="onClick"
     is-file="1"
     filepath="/file.txt"
    ></source>
</flow>
```

Figure 4.3: FlowDroid Analysis Result for *ReadFile.apk*



```
<flow>
    <sink method="&lt;java.io.FileOutputStream: void write(byte[])&gt;"
     is-file="1"
     filepath="/file.txt"
></sink>
    <source method="&lt;android.location.Location: double getLongitude()&gt;"
     in="getMyLocation"
     is-file="0"
    ></source>
    <source method="&lt;android.location.Location: double getLatitude()&gt;"
     in="getMyLocation"
     is-file="0"
    ></source>
    <source method="&lt;android.location.LocationManager: android.location.Location
    getLastKnownLocation(java.lang.String)&gt;"
     in="getMyLocation"
     is-file="0"
    ></source>
</flow>
```

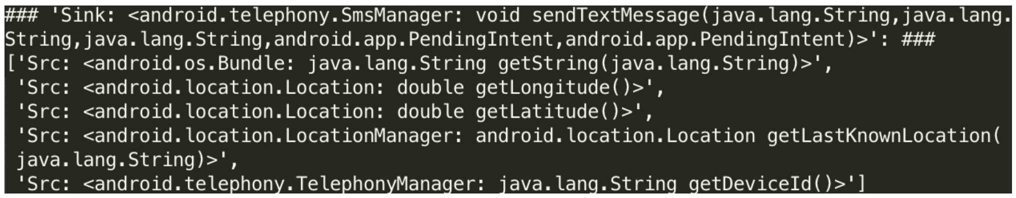
Figure 4.4: FlowDroid Analysis Result for *WriteFile.apk*

To verify the integrity of our implementation, we tested the enhancement with our example apps, *WriteFile.apk* and *ReadFile.apk*. The modified Phase 1 of DidFail changes the previous output of FlowDroid by adding two values, `is-file` and `filepath`, as shown in Figure 4.3 and Figure 4.4.

- The value `is-file` refers to whether this source or sink is related to the file system.
- The value `filepath` refers to the file path used by this source or sink.

In Figure 4.3, the file `file.txt` is used as a source via a call to `readLine()`. In Figure 4.4, the same file is used as a sink via a call to `write`. DidFail recognizes the inter-app taint flow, as shown in Figure 4.5.

Figure 4.5 shows five sources (including read from a location) and one sink (write to SMS), but no file-related sources or sinks for that taint flow. Figure 4.3 shows a read from location then write to file. Figure 4.4 shows a read from a file and then write to an SMS. Although the output from Phase 2 does not print flow information involving files, it still includes it in the analysis. This test shows that our implementation works and tracks file system taint more precisely than the previous version of DidFail.



```
### 'Sink: <android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.
String,java.lang.String,android.app.PendingIntent,android.app.PendingIntent)>': ###
['Src: <android.os.Bundle: java.lang.String getString(java.lang.String)>',
 'Src: <android.location.Location: double getLongitude()>',
 'Src: <android.location.Location: double getLatitude()>',
 'Src: <android.location.LocationManager: android.location.Location getLastKnownLocation(
 java.lang.String)>',
 'Src: <android.telephony.TelephonyManager: java.lang.String getDeviceId()>']
```

Figure 4.5: DidFail Analysis Result For Example App Set Containing Both *WriteFile.apk* and *ReadFile.apk*.

4.2 Tracking Taint of Content Providers

To improve support for Content Providers, we added the relevant methods of the ContentResolver class in Android to the file `SourcesAndSinks.txt`. This file is used by FlowDroid. It contains a list of methods with an extra annotation marking each method as either a source or a sink. This helps FlowDroid to generate intra-app flows. The methods we added are as follows:

- `ContentResolver.query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)`
- `ContentResolver.insert(Uri uri, ContentValues[] values)`
- `ContentResolver.update(Uri uri, ContentValues values, String where, String[] selectionArgs)`
- `ContentResolver.delete(Uri uri, String where, String[] selectionArgs)`

See the Android developer website [4] for a detailed description of these methods. To test this approach, we developed two example apps with different complexities. Each app defines its own Content Providers and has data flows that pass through them. We followed an incremental development approach. First, we added support by treating all Content Providers as a monolithic entity. Next, we added precision to our tracking mechanism by separating taints for each Content Provider. The two example apps test the two incremental development stages.

4.2.1 Treating All Content Providers as a Single Entity

The first example app consists of three components:

- A **Content Provider** that holds a list of text messages along with their sender information in a database.
- A **Broadcast Receiver** that listens for incoming text messages on the phone and inserts them into the Content Provider.
- A **main Activity** that reads one of the messages from the Content Provider and exfiltrates it out of the phone to a different recipient than the original sender of the text message.

The taint flow in this app (text received by the broadcast receiver → Content Provider → Read message and exfiltrate out) passes through the Content Provider. Our initial development efforts focused on marking this flow as tainted so that it is reported in DidFail's output. A visual representation of the flow is shown in Figure 4.6.

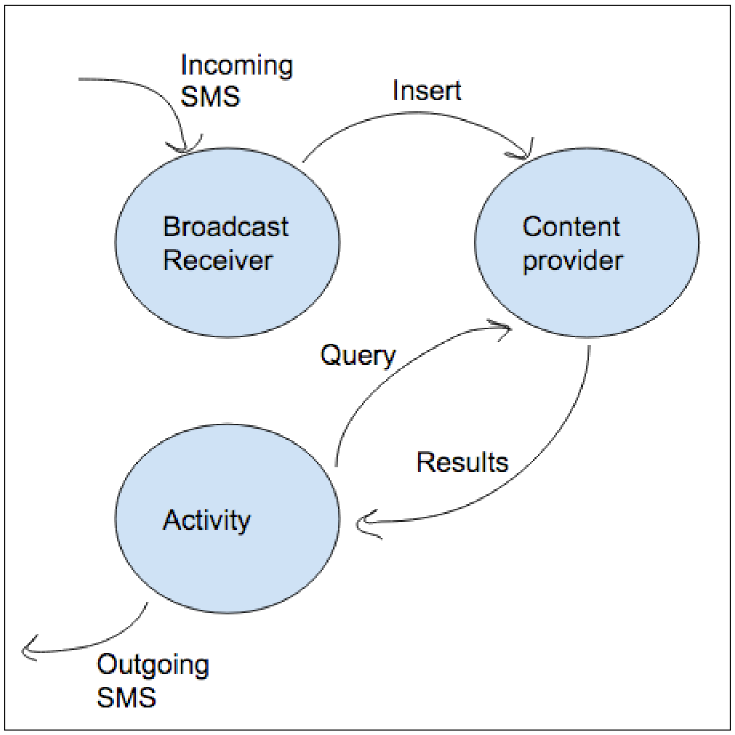
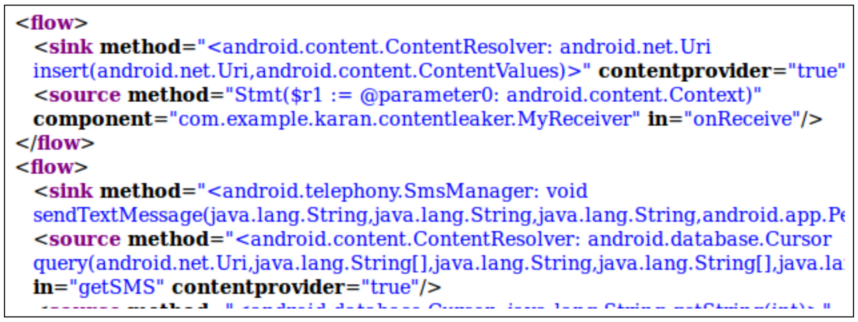


Figure 4.6: Flows in *Example App 1* for Content Providers

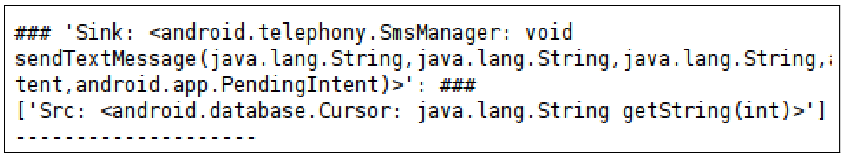
Once the relevant sources and sinks are listed in the file `SourcesAndSinks.txt`, flows that involve them appear in the FlowDroid output as shown in Figure 4.7. We made a slight change in FlowDroid to mark the flows related to the Content Provider with another attribute called `contentprovider` and set its value to true. DidFail makes use of this attribute during Phase 2 while processing these new flows.

During Phase 2, DidFail processes and connects these new flows; when appropriate, it marks them as tainted. After making changes to DidFail's Phase 2, we observed that DidFail can now connect the first source (receipt of the text message) and the final sink (exfiltrated SMS) as one flow through Content Providers, as shown in Figure 4.8.



```
<flow>
  <sink method="<android.content.ContentResolver: android.net.Uri
  insert(android.net.Uri,android.content.ContentValues)>" contentprovider="true"
  <source method="Stmt($r1 := @parameter0: android.content.Context)"
  component="com.example.karan.contentleaker.MyReceiver" in="onReceive"/>
</flow>
<flow>
  <sink method="<android.telephony.SmsManager: void
  sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app.Pe
  <source method="<android.content.ContentResolver: android.database.Cursor
  query(android.net.Uri,java.lang.String[],java.lang.String,java.lang.String[],java.la
  in="getSMS" contentprovider="true"/>
```

Figure 4.7: FlowDroid Output for *Example App 1* for Content Providers



```
### 'Sink: <android.telephony.SmsManager: void
sendTextMessage(java.lang.String,java.lang.String,java.lang.String,
tent,android.app.PendingIntent)>': ###
['Src: <android.database.Cursor: java.lang.String getString(int)>']
--------------------
```

Figure 4.8: DidFail Output Shows the Data Leakage via Content Providers for *Example App 1*

4.2.2 Improving the Precision of Taint Tracking

After we had a basic version of taint tracking working for Content Providers, we improved its precision. The previous example with *Example App 1* does track taints; however, it couples all Content Providers together into one entity. This implies that all flows are treated as if they are going to or coming from a single Content Provider. In real-world apps, however, this is rarely the case. To separate taints associated with each Content Provider, we uniquely mark each Content Provider with an identifier. The simplest form of a unique identifier that comes to mind for Content Providers is their content Uniform Resource Identifier (URI). Every Content Provider that is defined and implemented in Android is required to have a unique content URI. We added code to find the URI of a Content Provider when either of the Content Provider-related methods are found in the code. For example, if a method inserts data from an SMS into the Content Provider using a query method call, it will use the URI as the first parameter. We backtrack and find the URI if it is declared in the current function. Specifically, if the URI's declaration is something like

```
URI myuri = uri.parse("content://a/short/uri");
```

then our parsing code extracts out the declared URI `"content://a/short/uri"` and includes it along with the flow information in FlowDroid's output. We then use this additional information in Phase 2 of DidFail, combining Content Provider-related flows only if they have the same URI. In cases where our parsing code cannot find the URI (if it is not declared as above in the same function as a ContentResolver call such as `query()`), we use a fixed 64-byte string as the URI. This makes the Phase 2 code work irrespective of whether the URI was found in Phase 1. In the worst case, all taints of the Content Providers whose URIs are not found will be combined into one entity, similar to our initial implementation.

We developed another version of our example app with an additional Content Provider and Activity. This new Content Provider stores Global System for Mobile Communications (GSM) location information and the time at which the location was obtained. The newly added Activity has a method that obtains the current coordinates of the phone and inserts them into the Content Provider along with the time information. Another method in the same Activity reads location information from the Content Provider and logs it using the `Log.d`

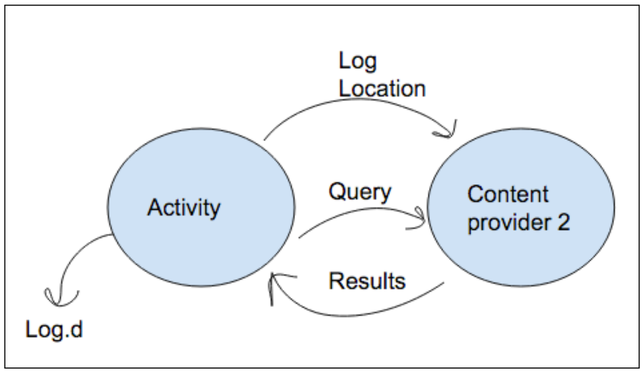
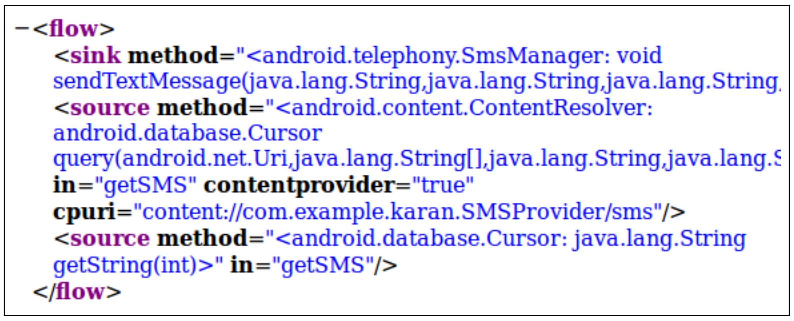


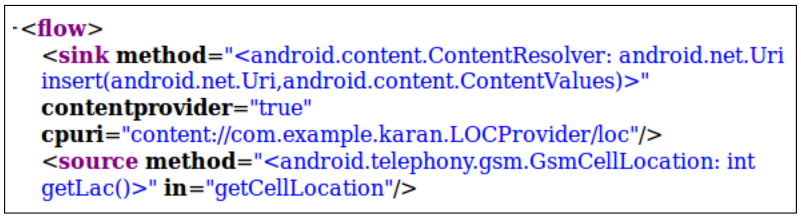
Figure 4.9: Flows in *Example App 2* for Content Providers

method. This effectively introduces a flow of sensitive information from reading the location to logging it via `Log.d`, with the Content Provider used for intermediate storage of the tainted data. These additional flows are shown in Figure 4.9. We see that the attribute called `cpuri` is added to the flows involving Content Providers. In Figure 4.10 and Figure 4.11, `cpuri=content://com.example.karan.SMSProvider/sms` is for the SMS Content Provider and `cpuri=content://com.example.karan.LOCProvider/loc` is for the Location Content Provider.



```
−<flow>
    <sink method="<android.telephony.SmsManager: void
    sendTextMessage(java.lang.String,java.lang.String,java.lang.String,
    <source method="<android.content.ContentResolver:
    android.database.Cursor
    query(android.net.Uri,java.lang.String[],java.lang.String,java.lang.S
    in="getSMS" contentprovider="true"
    cpuri="content://com.example.karan.SMSProvider/sms"/>
    <source method="<android.database.Cursor: java.lang.String
    getString(int)>" in="getSMS"/>
  </flow>
```

Figure 4.10: Flows in *Example App 2* for Content Providers



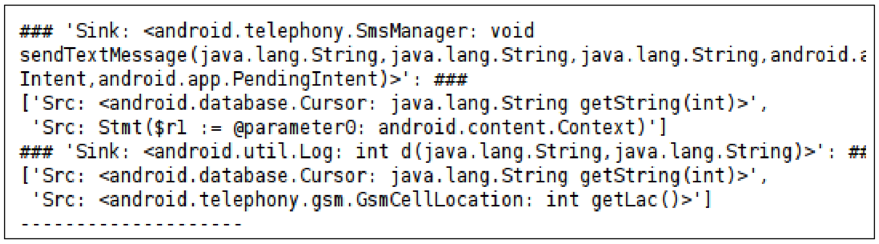
```
-<flow>
    <sink method="<android.content.ContentResolver: android.net.Uri
    insert(android.net.Uri,android.content.ContentValues)>"
    contentprovider="true"
    cpuri="content://com.example.karan.LOCProvider/loc"/>
    <source method="<android.telephony.gsm.GsmCellLocation: int
    getLac()>" in="getCellLocation"/>
```

Figure 4.11: Flows in *Example App 2* for Content Providers (continued)

The combined changes we made in Phase 1 and Phase 2 enable DidFail to precisely detect these flows. As we see in the DidFail output shown in Figure 4.12, flows of the SMS Content Provider are kept separate from those of the Location Content Provider.

4.3 Support for Dynamic Broadcast Receivers

As mentioned in Section 3, FlowDroid does not take dynamic receivers into account. DidFail therefore cannot directly use FlowDroid to detect taints involving dynamic broadcast receivers.



```
### 'Sink: <android.telephony.SmsManager: void
sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.a
Intent,android.app.PendingIntent)>': ###
['Src: <android.database.Cursor: java.lang.String getString(int)>',
 'Src: Stmt($r1 := @parameter0: android.content.Context)']
### 'Sink: <android.util.Log: int d(java.lang.String,java.lang.String)>': ##
['Src: <android.database.Cursor: java.lang.String getString(int)>',
 'Src: <android.telephony.gsm.GsmCellLocation: int getLac()>']
--------------------
```

Figure 4.12: Flows in *Example App 2* for Content Providers

However, the previous version of DidFail does track taints using a kludge. After a static analysis identifies a dynamically registered BroadcastReceiver, a static broadcast receiver can be manually added with a declaration in the manifest file. We automated the process of adding that declaration to the `manifest.xml` file in the latest version of DidFail.

The example code in Figures 4.13 and 4.14 shows how taint could be propagated via dynamic broadcast receivers.

```
this.recvr = new DeviceIDBroadcastAction();
registerReceiver(
  this.recvr,
  new IntentFilter("DeviceIDBroadcastAction"));
```

Figure 4.13: Code for *App A*

```
TelephonyManager tm = ...;
String tmDevice = tm.getDeviceId(); // A Source
Intent intent = new Intent();
intent.setAction("DeviceIDBroadcastAction");
intent.putExtra("deviceID", tmDevice);
sendBroadcast(intent); // A Sink
```

Figure 4.14: Code for *App B*

In Figure 4.13, *App A* registers the BroadcastReceiver dynamically using `registerReceiver()` and receives broadcast intents. *App B* has `READ_PHONE_STATE` permission; in Figure 4.14, it sends `deviceID` via broadcast intents. If *App A* is malicious, it can receive intents from *App B* that leak this sensitive data. *App A* would succeed in getting the broadcasts, as it registers the same Action string to receive the broadcasts. We created three example apps to simulate this behavior.

- *brecvDynamic*: Creates a dynamic receiver using `registerReceiver()` in `onCreate()` in `MainActivity`. Does not have any receiver declarations in manifest.
- *multipleBDR*: Creates dynamic receivers with different filters. Does not have any broadcast receiver declarations in manifest.
- *multipleBDR-static*: Same as *multipleBDR* but also has a receiver tag in manifest.

The rest of this section provides a high-level overview of the approach we took to solve this problem. We implemented a Python script that extracts data from Epicc and adds it to manifest files. Then, we modified Phase 1 of DidFail to run the new Python script before FlowDroid to detect flows that involve a dynamic broadcast receiver.

The new Python script calls the Epicc parser to get information about any broadcast receivers that might have been dynamically declared. As described in Section 1, Epicc provides a lot of information along with data about broadcast receivers. It provides information about sent intents, statically declared broadcast receivers, and their various tags. The Epicc output is parsed by the Python script, which looks for information about broadcast receivers. The script checks for multiple declarations of broadcast receivers and gathers all of the data in its tags. It uses this information to create XML broadcast receiver tags for dynamic receivers in the format required for statically declared receivers.

When Phase 1 of DidFail runs, it generates a manifest file in the output folder. An excerpt of an original manifest file is shown in Figure 4.15. The new script edits this manifest file, adding tags to it for dynamic broadcast receivers that can then be used by FlowDroid. An excerpt of the edited, augmented manifest file is shown in Figure 4.16.

```xml
<receiver android:exported="true" android:name="MyBroadcastReceiver">
    <intent-filter>
        <action android:name="DeviceIDBroadcast">
        </action>
    </intent-filter>
</receiver>
```

Figure 4.15: Taint Tracking: Original Manifest File

```xml
<receiver android:exported="true" android:name="MyBroadcastReceiver">
    <intent-filter>
        <action android:name="DeviceIDBroadcast">
        </action>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="DeviceIDBroadcastAction">
    <intent-filter>
        <action android:name="DeviceIDBroadcastAction"/>
    </intent-filter>
</receiver>
```

Figure 4.16: Taint tracking: Modified Manifest File

Figure 4.15 shows an example of the manifest file generated from Phase 1 of DidFail. We can see that it has a static receiver declared. This app also has a dynamic receiver declared in its `mainActivity` class where it creates a receiver that filters for action string `DeviceIDBroadcastAction`. Figure 4.16 shows the result after the new Python script parses Epicc's output and edits the manifest file. We can see that another receiver tag is added to the manifest file for the dynamic broadcast receiver.

Although the manifest file was created automatically, we did not automatically recreate the `apk` file; instead, we recreated it manually. Future work would be to automate this step. (Note that signing the `apk` file should not be important, since the modified `apk` file will only be analyzed by DidFail and not actually installed on a phone.)

4.4 Toggle for Computation-Intensive Analysis

We added a new argument to FlowDroid called `nostaticSourceSink`. If a user specifies this argument when running DidFail, the first phase will skip all static-field related analysis. This reduces DidFail's average memory requirements and analysis time.

5 Testing and Results

We tested our DidFail enhancements on 10 simple example apps that we developed specifically for testing. We also tested DidFail on more than 2000 real-world apps that we downloaded from the Google Play Store; these apps belong to many different Play Store categories.

Tests on taint tracking for file systems verified that our enhancement detects inter-app taint flows through files. For the example apps, our enhancement precisely identifies the inter-app taint flow and the file used for data flow between the two apps. From the real-world app set, we identified over 200 apps that read from or write to file systems. Due to time and resource constraints, we tested our enhanced analyzer on only 30 of these apps. Among those 30 apps, our analysis identified eight apps that contain file system-related sources or sinks. The total number of these sources and sinks is 119. Our implementation did not discover any inter-app taint flow among the 30 apps. The app set we tested is not comprehensive, so perhaps there are inter-app taint flows involving the file system in other sets of real-world apps. As of December 2016, 2.6 million apps were available for download from the Google Play store [12]. Also, we may have missed identifying some file-related taint flows because our method for obtaining the file path is based on static analysis. Because many apps generate file paths at runtime, the file paths we obtained might not be comprehensive.

We tested the changes for Content Providers on both our example apps and real-world apps. The code was correctly able to identify Content Provider URIs and fell back to combining taints if it was unable to parse the URI. Due to resource constraints, we restricted app size to 10 MB or less (FlowDroid analysis is especially resource and time-consuming for larger apps). Our initial analysis of the real-world app set identified 427 apps that were smaller than 10 MB and had Content Providers defined. We found fewer than 10 apps whose FlowDroid output contained flows related to Content Providers. Most of the real-world apps we tested that contained a Content Provider used it for directly reading or writing data, without using an implicit intent. An app can write to a Content Provider that another app reads from, enabling potential taint flows between apps, or it can write to a Content Provider that only it accesses [5]. However, the tests demonstrated that our enhancement identified flows related to Content Providers.

To verify our new functionality for dynamic receiver analysis in the Python script, we tested it on several example applications and real-world applications. In our tests, the script correctly parsed the output from Epicc and added the appropriate dynamically registered receivers as static receiver tags in the manifest file. FlowDroid then used this manifest file to detect taint flows involving these broadcast receivers. We tested the script on applications containing multiple dynamic receivers and others containing a single dynamic receiver. We error-tested using apps without dynamic receivers and verified that the script did not fail when run on these apps. We also performed manual testing to verify that the changes to the manifest created a functional manifest. We updated an `apk` file to include the newly-generated manifest file and verified that the updated `apk` file functioned.

5.1 Challenges and Future Work

The core limit to the new file system and Content Provider support is that the URI is hard to find in all cases unless it is hard-coded in the same function that makes read/write calls to the file system or Content Provider. In the future, we plan to add DidFail support for determining the value of dynamically defined URIs (e.g., URIs created by string concatenation or defined in a caller function) and URIs defined in a different file altogether. Constant propagation could be employed to find such URIs in the code, as done by the COAL solver [10]. Another current limit is that our implementation does not track flows directly between two Content Providers. For example, a flow in which a method of a Content Provider reads from and writes to another Content Provider will be missed by our analysis. We plan to add detection of those flows in a future version of DidFail.

Obtaining sufficient information about dynamic broadcast receivers was one of the biggest challenges we faced. Epicc does not specify the class that extends the BroadcastReceiver class. This class name is needed as a parameter for the receiver tag. As a workaround to this challenge, we obtained the class name from the declaration of static receivers. For cases where no static receivers were declared, we used random numbers as the class name so they could be distinguished.

Significant improvements can be made to DidFail's analysis of dynamically declared receivers to restrict potential data flows to control paths possible for flows reachable between dynamic registration and un-registration of a broadcast receiver (if it gets un-registered). DidFail currently performs a full taint flow analysis only for standard broadcast receivers. It needs further enhancements to perform analysis involving sticky and ordered broadcasts.

In the future, we also plan to test DidFail on more real-world apps to look for its shortcomings and improve upon them.

6 Conclusion

The previous version of DidFail did not support taint tracking through the file system, Content Providers, or dynamic receivers. It could not detect if a malicious app exfiltrated sensitive data using a taint flow traversing any of them. Also, the previous release had two separate branches. The first branch included analysis of static fields and on average required a lot of memory and longer compute times. The second branch contained other new functionality that on average performed faster and required less memory.

We added support for Content Providers and granular file system analysis by using a URI as the identifier for a unique source and/or sink in our taint tracking. We added support for dynamic broadcast receivers by performing an extra analytical step to find dynamic receivers and append descriptions of them to the manifest file before the data flow analysis. We also merged the functionality provided by the two branches of the previous DidFail release. We added a command line argument to enable users to disable static field analysis, reducing average memory requirements and analysis time.

With these new features added, DidFail identifies taint flows that it could not previously detect. Future improvements will build upon the current release of the software.

References

URLs are valid as of the publication date of this document.

[1] Jonathan Burket, Lori Flynn, Will Klieber, Jonathan Lim, Wei Shen, and William Snavely. Making DidFail Succeed: Enhancing the CERT Static Taint Analyzer for Android App Sets. Technical Report CMU/SEI-2015-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2015. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=434962.

[2] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[3] Android Developers. Android developers dashboard platform versions. https://developer.android.com/about/dashboards/index.html, January 2017.

[4] Android Developers. Android developers package index. https://developer.android.com/reference/packages.html, January 2017.

[5] Android Developers. Content providers. https://developer.android.com/guide/topics/providers/content-providers.html, February 2017.

[6] Lori Flynn and William Klieber. An enhanced tool for securing android apps. https://insights.sei.cmu.edu/sei_blog/2015/03/an-enhanced-tool-for-securing-android-apps.html, 2015.

[7] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, January 2014.

[8] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *International Workshop on the State Of the Art in Java Program analysis (SOAP)*, 2014.

[9] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *International Conference on Software Engineering (ICSE)*, 2015.

[10] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 77–88. IEEE Press, 2015.

[11] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android with *Epicc*: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, Washington, D.C., 2013.

[12] Statista. Number of available applications in the google play store from december 2009 to december 2016. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, December 2016.

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.		
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE July, 2017	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE DidFail: Coverage and Precision Enhancement		5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) Karan Dwivedi, Hongli Yin, Pranav Bagree, Xiaoxiao Tang, Lori Flynn, William Klieber, William Snavely		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2017-TR-007
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116		10. SPONSORING/MONITORING AGENCY REPORT NUMBER n/a
11. SUPPLEMENTARY NOTES		
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) This report describes recent enhancements to Droid Intent Data Flow Analysis for Information Leakage (DidFail), the CERT static taint analyzer for sets of Android apps. The enhancements are new analytical functionality for content providers, file accesses, and dynamic broadcast receivers. Previously, DidFail did not analyze taint flows involving ContentProvider components; however, now it analyzes taint flows involving all four types of Android components. The latest version of DidFail tracks taint flow across file access calls more precisely than it did in prior versions of the software. DidFail was also modified to handle dynamically declared BroadcastReceiver components in a fully automated way, by integrating it with a recent version of FlowDroid and working to fix remaining unanalyzed taint flows. Finally, a new command line argument optionally disables static field analysis in order to reduce DidFail's memory usage and analysis time. These new features make DidFail's taint tracking more precise (for files) and more comprehensive for dynamically declared BroadcastReceiver and ContentProvider components. We implemented the new features and tested them on example apps that we developed and on real-world apps from different categories in the Google Play app store.		
14. SUBJECT TERMS Android, taint, DidFail, mobile		15. NUMBER OF PAGES 25
16. PRICE CODE		

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500   Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102