Software Engineering Institute
Carnegie Mellon University

Making DidFail Succeed: Enhancing the CERT Static Taint Analyzer for Android App Sets

Jonathan Burket
Lori Flynn
William Klieber
Jonathan Lim
Wei Shen
William Snavely

March 2015

TECHNICAL REPORT
CMU/SEI-2015-TR-001

CERT Division

http://www.sei.cmu.edu

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 (Will Snavely, Jonathan Burket, Jonathan Lim, and Wei Shen) to work on this project. We thank Dr. Alexandre Bartel of the Technical University of Darmstadt, Germany, for his useful discussion about FlowDroid developments for dynamically registered BroadcastReceivers and details of its `nopath` option. Lori Flynn and Will Klieber worked on this project thanks to funding by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon (for the operation of the Software Engineering Institute (SEI), an FFRDC), by the Office of Naval Research (ONR) grant N000141310156.

Abstract

This report describes recent significant enhancements to DidFail (Droid Intent Data Flow Analysis for Information Leakage), the CERT static taint analyzer for sets of Android apps. In addition to improving the analyzer itself, the enhancements include a new testing framework, new test apps, and test results. A framework for testing the DidFail analyzer, including a setup for cloud-based testing was developed and instrumented to measure performance. Cloud-based testing enables the parallel use of powerful, commercially available virtual machines to speed up testing. DidFail was also modified to use the most current version of FlowDroid and Soot, increasing its success rate from 18% to 68% on our test set of real-world apps. Analytical features were added for more types of components and shared static fields and new apps developed to test these features. The improved DidFail analyzer and the cloud-based testing framework were used to test the new apps and additional apps from the Google Play store.

1 Introduction

DidFail (Droid Intent Data Flow Analysis for Information Leakage) is a static analyzer, 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. This technical report describes enhancements recently made to the analyzer itself, a new testing framework, newly developed test apps, and test results.

The permission system is an important feature of Android security. By placing adjustable limitations on what resources applications can use, Android allows users a large degree of control over how much trust should be given to specific applications. This system allows untrusted applications to coexist in an environment with sensitive information, such as the user's location, contacts, and other personal details.

One of the core assumptions of the Android permission system is that an application that lacks permission to access a resource cannot obtain access to that resource. This assumption is enforced with code that checks that an application has permission for a resource when it requests it from the system. Once the application has access to the resource, however, it can use that resource as it sees fit. This ability to freely use the resource can include sharing private information with other applications or executing actions on behalf of other applications. Indeed, while the Android permission system is well suited for controlling access for applications in isolation, it can fail to protect resources on the device.

Consider the case where a user installs an application for recording audio notes. Because the user is recording sensitive information, he or she verifies that the application has no network access. The same user then installs a weather application that uses the internet but has access to no sensitive content. Considering the applications in isolation, the user has made sensible decisions; the application that has internet access does not have access to sensitive information and vice versa. If, however, these applications can communicate (either intentionally or unintentionally), then the user's information may be at risk. If the audio recording application can send messages to the weather app that are then sent over the internet, an outside attacker could gain access to audio recorded by the device.

Considering the Android permission system alone is insufficient for reasoning about scenarios where information flow between applications circumvents intended permission limitations. *DidFail* is a tool for detecting such flows [1]. DidFail statically analyzes sets of Android application packages (APKs) to detect cases where sensitive information may be leaked from one application to another. While DidFail is a useful tool, it previously fell short in several scenarios. In its previous release, DidFail tracked flows between Activities only by intents. Other types of dataflows, such as those through static fields or between an Activity and a BroadcastReceiver, were missed. With the work detailed in this report, we expanded DidFail's capabilities to work with more applications and handle more types of flows.[1] In sections 3 through 6, we detail four improved DidFail features.

The main DidFail webpage is `http://www.cert.org/secure-coding/tools/didfail.cfm`. The version of DidFail and the test apps described in this report are also directly available on the Carnegie Mellon University website:
`https://www.cs.cmu.edu/~wklieber/didfail/install-latest.html`.

[1] Some flow types are yet to be implemented.

2 DidFail

DidFail is a static analysis tool for Android applications that detects possible information flows between applications. DidFail works using two phases.

In Phase 1 (Figure 2.1), 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 [2] and Epicc [3].

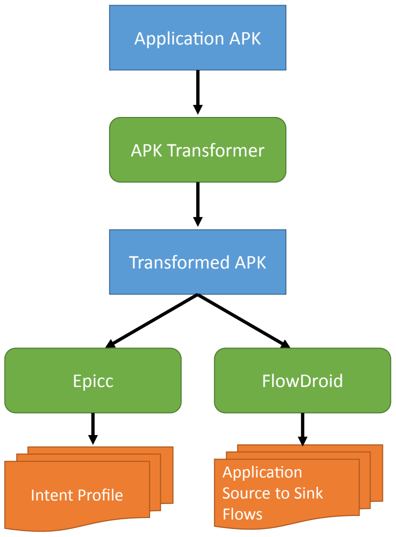

Figure 2.1: Overview of DidFail Phase 1

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, reporting all flows from sensitive sources to sinks. Sources include function calls that access sensitive information in Android, such as `getLatitude()` or `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. Output from FlowDroid might identify, for example, that an application reads contact information from some source, then sends it as part of an intent, or that an application reads information from an intent, then sends it as a Short Message Service (SMS) message.

Epicc [3] 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 what 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 and, once completed, does not need to be run again.

Phase 2 of FlowDroid combines the Phase 1 output of multiple applications to determine how specific applications in a set can interact. Consider our example from Section 1 of the audio and weather applications. If these applications colluded via an intent, then in Phase 1, DidFail would discover that the audio app obtains audio information (a taint source) and the information flows to an intent sent to an application with a specific name.

Similarly, Phase 1 would reveal that the weather app handles intents, reads information from the intent, and then sends it over the internet.

In Phase 2, DidFail combines the information about these two applications. DidFail analyzes dataflows between apps that can occur through intents, eventually discovering and reporting that sensitive audio information is sent over the internet.

Cases like this one are where DidFail shines. Unfortunately, the tool is still limited in the types of flows it can analyze. In our work, we enhanced DidFail to handle more applications and more types of flows.

3 Flows Involving Services

A Service is an application component that is meant to run without direct user interaction (e.g., provide some functionality while the app is not in the foreground). A Service can be instantiated by passing an intent corresponding to a Service's intent filter, or the name of the class instance, and making a call to the `startService` method. We added new functionality to DidFail to handle some dataflows involving Services.

3.1 General Modifications

The following sections describe the general modifications we made to the DidFail tool.

3.1.1 Recognizing New Intent Sinks

DidFail's *APK Transformer* identifies intent sinks in an Android app by matching method signatures in the app with known intent sink method signatures. If the signatures match, an intent ID is generated for the intent sink method. Method signatures for the non-Activity intent sinks were added to track intent sinks for BroadcastReceivers and Services. Similarly, FlowDroid had to be modified slightly to recognize the new intent sinks.

3.1.2 Matching Flows by Component Types

DidFail initially handled only flows involving activities. Extending DidFail to also support BroadcastReceiver and Service components required a slight modification to how the sources and sinks were matched. Certain methods indicate the type of a receiving component and need to be accounted for when matching receiving components and sending components (*e.g.*, *startService* → *service*, *startActivity* → *activity*). By recording the sink component type, we enabled the matching process to create appropriate flows.

3.2 Initial Test Results

Using two apps we created, we tested the analyzer on a single app set. *SendApp*'s main Activity component obtains location information by calling `getLocation()` and stores the information in an intent set with a custom action string matching *SendAppService*'s intent filter.

As illustrated in Figure 3.1, the intent is passed to `startService`, received by *SendApp*'s Service component, and similarly sent to the *ReceiveAppService* component. *ReceiveAppService* passes the tainted information to *ReceiveApp*'s Activity component by calling `startActivity`.

Phase 1 and Phase 2 analysis results were as we expected, tracing the tainted location information through the Activity and Service components as well as identifying the sources and sinks. We also ran a preliminary test of 100 APKs for Phase 1 using the extended APK Transformer, Epicc, and FlowDroid. We observed that intent sinks, such as `startService` and `bindService`, were identified. Further testing still needs to be done to verify that the modifications improve Phase 2 results.

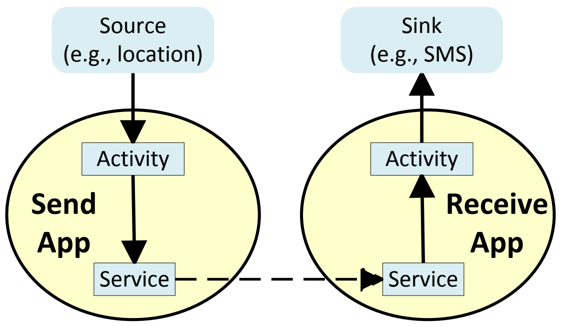

Figure 3.1: Initial App Set for Service Components

4 Flows Involving BroadcastReceivers

Previous versions of DidFail failed to detect dataflows that involve broadcast intents. Broadcast intents are received in a particular way; the app contains a class that extends the class `android.content.BroadcastReceiver`, which implements the `onReceive` method. An application can declare the broadcasts it wishes to receive statically in the manifest or dynamically with the `registerReceiver()` method. There are different types of broadcasts: normal broadcasts with unordered receivers, ordered broadcasts, and sticky broadcasts that remain active after the initial broadcast distribution is completed.

An app can unregister a dynamically registered receiver (and thereby stop it from receiving broadcasts) by calling the `Context.unregisterReceiver()` method. DidFail does not consider calls to `unregisterReceiver()`; if it can be proven that a receiver cannot be live when a tainted broadcast is sent, then DidFail produces a false alarm.

We made three modifications to allow DidFail to analyze dataflows that include broadcast intents. As mentioned in Section 4.6, a manual adjustment is necessary to handle dynamically registered receivers, and we do not yet analyze for the dataflow of sticky broadcasts and ordered broadcasts. We enhanced the Epicc parser to extract information about broadcast intents, modified FlowDroid to mark broadcast-related functions as sources or sinks, and extended Phase 2 to consume the new output resulting from the previous modifications.

4.1 Parsing Epicc Output for BroadcastReceivers

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`." Potential filter properties are given as a list (for example, the Actions field in Figure 4.1).

```
Type: android.content.BroadcastReceiver
No permission
Intent Filter value: 1 possible value(s):
Actions: [DeviceIdBroadcast]
```

Figure 4.1: Example Epicc Output for a BroadcastReceiver

We added a separate parser function named `parse_bcast` to address this issue. It is used by DidFail in addition to the previously existing parser function named `epicc_parser`.

4.2 Marking Broadcast Intents as a Sink/Source in FlowDroid

Function signatures related to transmitting and receiving broadcast intents were added to the APK Transformer and FlowDroid so that they were recognized as sources or sinks. Table 4.1 shows the related functions.

Table 4.1: Functions Related to Broadcasting Intents

`onReceive`
`sendBroadcast`
`sendBroadcastAsUser`
`sendOrderedBroadcast`
`sendOrderedBroadcastAsUser`
`sendStickyBroadcast`
`sendStickyBroadcastAsUser`
`sendStickyOrderedBroadcast`
`sendStickyOrderedBroadcastAsUser`

4.3 Trace Broadcast Flows in Phase 2

The `taintflows.py` program (responsible for executing Phase 2 of DidFail) was extended to parse the additional sinks and sources introduced in the previous sections and to match broadcast intent filters. The logic is the same as with non-broadcast intents.

4.4 Test Applications

We wrote four apps to test this new feature:

- *BroadcastTx* has the READ_PHONE_STATE permission and sends the device ID via a broadcast intent with an action filter. See Figure 4.2 for the code that leaks the device ID.
- *BroadcastRx_Static* has the SEND_SMS permission and receives broadcast intents via a statically declared receiver with the same action filter as declared in the manifest file. See Figure 4.3 for the manifest, and Figure 4.4 for the receiver class.
- *BroadcastRx_Dyn* has the ACCESS_FINE_LOCATION permission and receives a broadcast intent via a dynamically registered receiver with the same filter. See Figure 4.6 for the receiver registration, and see Figure 4.7 for the receiver class.
- *KludgeBroadcastRxDyn* is the same as *BroadcastRx_Dyn*, except it has a filter declaration in the manifest (see Figure 4.5) as well as a standard dynamic registration of the BroadcastReceiver in the component's Java code.

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

Figure 4.2: Transmitter Code Leaking Device ID Through a Broadcast Intent

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

Figure 4.3: Statically Declared BroadcastReceiver in a Manifest File

```
public void onReceive(Context context, Intent intent) {
    Bundle extras = intent.getExtras();
    SmsManager smsManager = ...;
    String number = "55555555555";
    String id = extras.get("deviceID");
    smsManager.sendTextMessage(number, null, id, null, null);
}
```

Figure 4.4: Code for a BroadcastReceiver that Sends Text Messages

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

Figure 4.5: Manifest Code of *KludgeBroadcastRxDyn*, with Manually Added Filter for Dynamically Registered BroadcastReceiver

4.5 Results

Our modified FlowDroid identified the expected flows from sources to sinks in three of the four toy applications: *BroadcastTx*, *BroadcastRx Static*, and *KludgeBroadcastRxDyn*. For these apps, it identified the leakage of the device ID through a broadcast intent in Figure 4.2 as a tainted source. The BroadcastReceivers in Figure 4.4 and Figure 4.7 were identified as sinks.

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

Figure 4.6: Registering a BroadcastReceiver Dynamically

```
public class MyBroadcastReceiver_Dyn extends BroadcastReceiver {
    public void onReceive(Context c, Intent i) {
        String file = "devicelDSinkFile.txt";
        Bundle extras = i.getExtras();
        String id = extras.get("deviceID");
        // Write id to the file...
    }
}
```

Figure 4.7: Code for a BroadcastReceiver that Writes to a File

Phase 2 analysis correctly identified two inter-application flows: (1) from the transmitter to the statically registered BroadcastReceiver and (2) from the transmitter to the dynamically registered BroadcastReceiver.

DidFail did not detect the dataflows from sources to sinks in the other app, *BroadcastRx_Dyn*, for reasons described in Section 4.6.

Because FlowDroid was unable to statically detect flows in *BroadcastRx_Dyn*, we tested this app on a physical phone, alongside *BroadcastTx*. Figure 4.8 shows the log messages (from `adb logcat`) when running *BroadcastTx* and *BroadcastRx_Dyn*.

```
D/BcastTx (28361): OnClick
D/BcastTx (28361): DeviceID: 35[redacted]03
D/BcastTx (28361): Intent has been sent.
D/BcastRxDyn(28399): 35[redacted]03
```

Figure 4.8: Output of `adb logcat` (The Device ID Has Been Partially Redacted.)

4.6 Dynamically Registered BroadcastReceivers

Currently, for a dynamically registered BroadcastReceiver to be handled by DidFail, a dummy static declaration of the BroadcastReceiver must be added to the manifest file so that FlowDroid can appropriately analyze it. This dummy declaration can be added after analysis of the code shows a BroadcastReceiver is dynamically registered, as successfully demonstrated in the test app *KludgeBroadcastRxDyn*.[1]

[1]This and the other test apps we developed for this project are available for free download at: http://www.cs.cmu.edu/~wklieber/didfail/test-apps-dec2014/

Some problems with analyzing dynamically registered BroadcastReceivers have been fixed in the latest version of FlowDroid's development branch. The develop branch of the standard FlowDroid distribution[2] has newly added handling of some dataflows involving dynamically declared BroadcastReceivers. Their DroidBench[3] test app distribution now includes two apps (*BroadcastReceiverLifecycle2* and *BroadcastTaintAndLeak1*), which have dynamically declared BroadcastReceivers with taint flows that FlowDroid detects, and it detects taint flow in our test app *KludgeBroadcastRxDyn*. However, at the time of this writing, the taint flow in our test app *BroadcastRx_Dyn* is not detected by FlowDroid.

4.7 Future Work

We plan to modify DidFail to handle dynamically declared BroadcastReceivers in a fully automated way by integrating it with a recent version of FlowDroid and working to fix remaining un-analyzed taint flows. We plan to investigate modifying FlowDroid to analyze the flows, or alternatively, modifying the APK Transformer to automatically add this dummy declaration into the app's manifest file. We also plan to add analysis specific to ordered and sticky broadcasts.

[2]https://github.com/secure-software-engineering/soot-infoflow-android
[3]https://github.com/secure-software-engineering/DroidBench/tree/develop

5 Flows Involving Static Fields

There are many ways that Android apps could communicate. The most direct way is for applications to send and receive intents, which are specifically designed for inter-application communication. Other, more subtle communication mechanisms can also be used to share information between applications, including cooperative use of shared state, or covert channels such as power usage and timing. DidFail needs to understand as many communication channels as possible, to determine how applications share permissions.

5.1 The Issue with Static Fields

While DidFail does not detect many of these communication channels, support was successfully added for detecting some flows through *static shared fields*. A static field in Java (denoted by using the `static` keyword) shares its value among all instances of the class. The declaration of static fields establishes state at the class level rather than at the instance level.

Static fields present an interesting challenge when trying to detect inter-application flows on Android. Consider a hypothetical application, *AppChat*, that receives intents, appends the contents of received intents to a static String `holdover`, and then returns the contents of `holdover` to the application that originally issued the intent. This *AppChat* application allows two applications (A and B) to communicate with each other via proxy. A can send a message to *AppChat*; that message is saved in the static field `chatlog`. When B sends a message to *AppChat*, it will receive back the message sent to *AppChat* by A. Consequently, in the application set containing A, B, and *AppChat*, there exist six potential inter-application flows: $A \to AppChat$, $AppChat \to A$, $B \to AppChat$, $AppChat \to B$, $A \to B$, and $B \to A$.

Unfortunately, DidFail originally failed to detect some of these potential flows, due to its use of the FlowDroid tool [2]. FlowDroid statically detects potential flows from labeled source and sink functions in an APK. In this case, `getIntent` is considered a source, while `setResult` is considered a sink. Consequently, FlowDroid reports a single flow from `getIntent` to `setResult`. Notably, this result is the same as if the static field `holdover` had not been used at all (that is, the received message is echoed back as the reply). Indeed, while FlowDroid does track through static fields, it does not indicate their presence in the results. As a result, the second phase of DidFail is unaware of the presence of any static fields in an application.

```
static String log = "";

private void getDataFromIntent() {
    Intent i = getIntent();
    String message = getString("message");
    log = log + message;
    Intent reply = new Intent();
    reply.putExtra("STORED_DATA", log);
    setResult(Activity.RESULT_OK, reply);
}
```

Figure 5.1: Simple Static Field Example (Simplified Version of AppChat Code)

Given only the flow from `getIntent` to `setResult`, DidFail understood that the *AppChat* application accepts input from an application and then replies to that application based on that input. As a result, DidFail reported the following flows: $I_A \to R(I_A)$ and $I_B \to R(I_B)$. These flows account for four of the six of the flows we described above ($A \to AppChat$, $AppChat \to A$, $B \to AppChat$, $AppChat \to B$). Crucially missing, however, was the flow between applications A and B, which can communicate via the static field `holdover`.

5.2 Adding Support for Static Field Flows

For DidFail to analyze flows involving static fields, FlowDroid needed to be modified to report about how static fields are used in an application. Given that static fields provide persistent state that can be both read from and written to, it makes sense to treat static fields as both a source *and* a sink. In other words, if sensitive information flows from a source to a static field, we should report the flow, and if content flows from a static field to a sink, we should report that flow as well.

There were a number of key challenges we encountered while trying to add this feature to FlowDroid. First and foremost, FlowDroid handles only sources and sinks that occur as function calls. Consequently, while it is easy to mark a function as a new taint source, there is no support for adding more general types of sources, such as those associated with a specific variable. This limitation meant that we needed to modify the actual code associated with FlowDroid's taint analysis to achieve our goals.

At a high level, FlowDroid discovers flows in an application using the following steps:

1. Load source code of the application, convert the code to Jimple [4], and construct an *exploded supergraph* that models interprocedural interactions. Note that a node is a statement in Jimple, and an edge is directed, pointing from a node A to B, if B can succeed A in some possible program execution. Each node also has a taint status T.
2. Scan through the graph and add an edge ($\emptyset$, N) for every N that contains a source to a worklist W.
3. While W is non-empty, pop edge (N_1, N_2) from W:
   (a) Given the current taint status on N_1, run the *flow function* on (N_1, N_2) to produce the current taint status for N_2.
   (b) If the taint status of N_2 changed, for every child of N_2, N_3, add edge (N_2, N_3) to the worklist W.
4. Use the per-statement taint information to compute the source-to-sink flows through the application.

The majority of this algorithm is based on the general IFDS Framework for dataflow analysis [5]. The key component of FlowDroid's instantiation are the *flow functions* used to propagate taint information across edges. These flow functions, one of which is selected based on the type of statement in question, dictate when taint information is created, propagated, and destroyed. Consider an assignment statement $x = y$. If y is tainted, then we should mark x as tainted, and if y is not tainted, then we do not mark x as tainted.

In the original FlowDroid, because all taint sources and sinks are function calls, an assignment of the form $x = y$ can only propagate taint, not create or destroy it. To support static fields, however, this approach must be changed. We modified the flow function for assignment statements of the form $x = y$ to instead have the following logic:

- If y is a static field, mark x with taint T_y, where T_y is taint originating from static field y.

```
1  class Box {
2      String x;
3  }
4
5  static Box b = new Box();
6  ...
7  Box b2 = b;
8  b2.x = source();
9  sink(b2.x);
```

Figure 5.2: Static Fields with Aliasing

- If x is a static field and y is tainted with taint T, add a $T \rightarrow x$ to the list of sink results.
- Otherwise, the taint of x is set to the taint of y.

In addition, we altered the original code that seeds the IFDS algorithm with initial edges to also add edges for assignment statements involving reads from static fields. This modification works quite well for basic examples. FlowDroid now correctly reports the flows *getIntent* $\rightarrow$ *holdover*, *holdover* $\rightarrow$ *setResult*; and *getIntent* $\rightarrow$ *setResult* from the Figure 5.1 example. With the static field information now recorded as extra information flows, judgments can now be made about inter-application flows through static fields. (See Section 5.4.)

5.3 Harder Static Field Flows

Unfortunately, this simple solution does not work in a non-trivial case. Consider the situation where a static variable has been *aliased*, as in Figure 5.2. In this case, when we run the flow function on the line $b2.x = source()$, we should treat $b2.x$ as a sink because it is a static field. It is not obvious that term $b2.x$ is a static field, however. It is only the fact that $b2.x$ comes after the line $b2 = b$ that causes it to be a static field. What this reveals is that, in the general case, *context is necessary* for identifying static fields. If $b2.x$ occurred after $b2 = newBox()$, then $b2.x$ would not be considered to be a sink on Line 8.

Naturally, whether a given term $x.y.z... = t$ is a write to a static field is an undecidable property. We can, however, obtain a conservative guess of which fields are static by creating an additional FlowDroid pass that occurs before the main taint analysis. This pass, which uses the same type of taint analysis, marks all static variables as sources and then propagates taint forward. For the code in Figure 5.2, b would be tainted on Line 7, which would cause $b2$ to be tainted. Because $b2$ is tainted, $b2.x$ is also tainted. Once we finish with this initial pass, we record the results of what is tainted in a set.

With this initial pass completed, we now have enough information to solve our aliasing problem. We no longer need to guess whether $b2.x$ is a static field; we can simply check whether it is in the set of statements marked as tainted in the first pass. With this modification, our flow function for assignment $y = x$ now becomes the following:

- If y is a static field, mark x with taint T_x, where T_x is taint originating from static field x.
- If x is a static field **OR** x **is in the set of tainted statements from Pass 1**, and y is tainted with taint T, add a $T \rightarrow x$ to the list of sink results.
- Otherwise, the taint of x is set to the taint of y.

This reuse of FlowDroid's taint analysis to find static fields turns out to be a pleasant and concise solution. In contrast, our initial attempts at a solution in one pass that involved backtracking were much more complex and did not generalize well.

5.4 Phase II Static Field Analysis

Phase II still needs to make use of the flows through static fields information produced in Phase I. Recall that in our original example involving applications A, B, and *AppChat*, DidFail found the flows $I_A \rightarrow R(I_A)$ and $I_B \rightarrow R(I_B)$, where I_A and I_B are intents issued by A and B, respectively.

When we run modified FlowDroid on *AppChat*, we obtain the flows *getIntent* → *holdover*, *holdover* → *setResult*, and *getIntent* → *setResult*, as stated earlier. Because *holdover* is a static field that could persist across the handling of multiple intents, *every application* that reads from *holdover* is tainted by *every application* that writes to that static field. This approach effectively amounts to taking the Cartesian product of all reads and writes to the field. We find that application A and B both write to and read from *holdover*, so taking the Cartesian product, we get $(A, A), (A, B), (B, A)$, and (B, B). This result means that in addition to the flow where A and B interact with *AppChat* in isolation, we must now consider the case where input from A flows to B and vice versa.

These additional flows from the Cartesian product calculation are added to the set of flows after DidFail generates the Phase II flow equations. (Generating the Phase II flow equations is described in detail in Section 3.3 of [1]).

5.5 Results

Our enhanced DidFail identified the expected flows from sources to sinks in a set of three toy apps we developed, which provided the taint flow described in sections 5.1 and 5.4: *AppChat*, *SendSMS2*, and *LogWrite*. *SendSMS2* calls source method `getLine1Number()`, which gets the MSISDN for a GSM phone. *SendSMS2* sends data it receives from *AppChat* into an SMS-sending sink call. *LogWrite* calls source method `getLastKnownLocation(java.lang.String)` and sends data it receives from *AppChat* into a sink method call to `Log.w`. DidFail finds that both sources can reach both sinks via data flows that involve a static field in *AppChat*.

Figure 5.3 shows an example static field setup where FlowDroid fails to identify any flow through the static field, and so our modifications also fail to trace taint through it. The taint flow analysis fails to detect flows, for flows involving fields of static classes that another class gets and sets.

5.6 Drawbacks, Limitations, and Future Work

Our analysis appeared to work correctly, except for fields of static classes that another class gets and sets, as discussed in 5.5. We plan to modify FlowDroid to detect those taint flows. While our modifications to support flows through static fields worked quite well in most cases, there are some additional issues with our solution. With two taint passes instead of one, that analysis can take twice as long as the original analysis.

More critically, the FlowDroid runtime also scales with the number of sources, which means that adding a new source for every static field read could also severely impair performance depending on how frequently static fields are used. To evaluate how important this issue is, we tested 100 APKs that, prior to adding this new feature, were successfully analyzed within the allocated time and memory (using virtual machines with 61 GB and FlowDroid analysis memory allowed

```
public static class staticBox {
    public String holdoverFieldA = "";
}

private static staticBox holdover = new staticBox();

public class holdingBox {
    public String getStatic() {
        return holdover.holdoverFieldA;
    }
    public void setStatic(String s) {
        holdover.holdoverFieldA = s;
    }
}
```

Figure 5.3: Field of Static Class; Other Class Gets and Sets Field

to take up to 32 GB). (See Section 6.) Unfortunately, the majority (66 out of 100) timed out or ran out of heap space. Most applications make frequent use of static fields, particularly when invoking libraries such as Google Ads. For many of these applications, an additional taint analysis for every static field read was prohibitively expensive. In the future, we will increase virtual machine memory size to see if it enables successful static field taint analysis of more apps. Virtual machines with 244 GB of memory are available,[1] which also have 32 virtual cores. Future work will also develop strategies to reduce the number of reads that get traced through the program.

As noted earlier, detecting which fields are static is undecidable and our taint-based solution is an overapproximation. Consequently, some reported flows to static fields may not be realizable in practice. The solution is not a *strict* over approximation, however, because there are some cases where FlowDroid itself does not recognize that a flow exists at all. We cannot detect flows that use static fields for legitimate flows missed by FlowDroid.

Finally, the current implementation does not support static fields involving arrays. This feature should be straightforward to add, however.

6 Infrastructure Improvements

The original DidFail implementation experienced a high failure rate in the Phase 1 analysis. Recall that Phase 1 analysis is run on each APK separately to find tainted flows in these apps. This first step of the analysis transforms the input APK, instrumenting the callsite of intent-sending methods with a unique ID. The transformed APK is then processed by FlowDroid and Epicc. With this improvement, we sought to increase the success rate of APK processing in Phase 1.

[1] http://aws.amazon.com/ec2/pricing/

6.1 The Problem

We identified several issues that contributed to a high failure rate in Phase 1. The APK Transformer and FlowDroid rely on the underlying Soot framework to handle reading and writing APK files. While the APK Transformer itself rarely failed, FlowDroid often choked on the APK produced by the APK Transformer. Moreover, some APKs encountered no explicit FlowDroid failures, but were not analyzable in a timely fashion (the FlowDroid invocation ran out of time or memory). These issues together resulted in a very low success rate in Phase 1. There were no significant issues encountered with the Epicc tool. In the time since the original DidFail version was completed in March 2014, a new version of the Soot framework was released. This new version included significantly better facilities for handling APK files, but it was not completely compatible with code written against the older version of Soot. A new version of FlowDroid was also released, which used the new version of Soot.

6.2 Porting Phase 1

Our first step was to port the DidFail changes to the latest versions of Soot and FlowDroid. This step involved inspecting the modifications to the old FlowDroid branch, moving them to the latest FlowDroid branch, and then testing for correctness. Moreover, the APK Transformer was changed to use the latest version of Soot, requiring a few source changes and further testing.

6.3 Measuring Phase 1 Performance

The next step was to create an instrumented harness for running the various components of Phase 1. In particular, we were interested in gathering data about FlowDroid, since in previous test runs many APKs timed out during FlowDroid processing. A small test harness was developed in Python for running various Phase 1 components (the APK Transformer, Epicc, FlowDroid, etc.) with various parameters and with a specified timeout. This harness also measured system metrics (e.g., memory usage, CPU usage) while the DidFail analyses were running.

6.4 Initial Testing

A selection of 90 APKs was chosen randomly from a large set, with the goal of running them through both the old and new versions of DidFail to evaluate the effectiveness of our changes. A machine was provisioned from Amazon EC2 to run the test because it was difficult to secure time on a machine locally with enough resources. This machine was equipped with 4 cores and 32 GB of memory. FlowDroid was allocated 16 GB of memory (through the heap space parameter to the JVM) and a 15-minute timeout. APKs were processed by the APK Transformer before being sent to FlowDroid. The graph in Figure 6.1 summarizes the results of this test.

The number of APKs successfully processed by FlowDroid roughly tripled from 16 to 61 with very few failures. A few APKs were skipped due to errors in the transformation stage. There were 21 APKs that timed out during FlowDroid analysis—5 more than timed out in the initial run. The smaller number of timeouts in the initial run may be explained by the high failure rate; if more APKs hadn't run into runtime exceptions, they may have eventually timed out. This test run showed that our modifications substantially improved the baseline success rate of Phase 1 analysis.

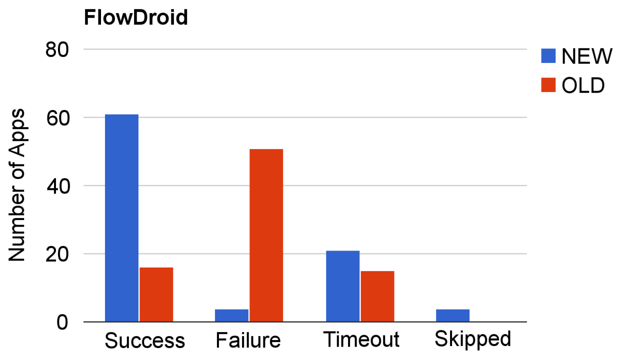

Figure 6.1: Results of FlowDroid Test Run, With and Without Modifications

6.5 FlowDroid Performance Tweaks

Next, we investigated the 21 timeouts in the FlowDroid test. By reviewing the performance counters, we saw that these analyses consumed all the memory allocated to them (16 GB) and were unable to make further progress.

The first thing we tried was vertical scaling. We provisioned a much larger machine and ran FlowDroid again on one of the timed-out APKs. This run was not productive. Despite allocating nearly 170GB of memory to the process and disabling timeouts, the analysis reached the same state as before: consuming all allocated memory and making next to no progress. Figure 6.2 shows memory usage over time (approximately 2 hours) for that analysis attempt.

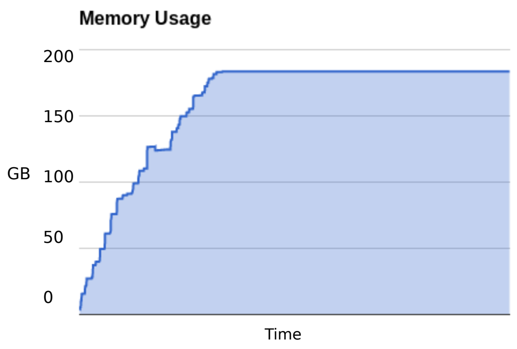

Figure 6.2: Results of the FlowDroid Test Run on a High-Performance Machine

It was clear that the analysis itself needed to be tuned. Recall that FlowDroid seeks to construct paths between sources and sinks in an application. Two fundamental aspects we could tweak were the source/sink identification process and the pathfinding process. Regarding the former, we could try to reduce the number of sources and sinks by filtering them in some fashion. Regarding the latter, we could reduce the precision of the pathfinding process. Of course, this tuning impacts the quality of the analysis. Filtering the sources/sinks may cause FlowDroid to miss a potentially dangerous flow. Dumbing down the pathfinding bears this risk as well, and carries the additional risk of introducing false positives.

Nonetheless, we were interested in determining how much tuning FlowDroid required to process the remaining 21 APKs. We provisioned the same machine as before (32 GB of memory, 4 cores), and maintained the same timeout (15 minutes).

Source/sink filtering resulted in 9 out of the 21 APKs getting successfully processed by FlowDroid. Source/sink filtering uses the FlowDroid command-line parameters `nocallbacks` and `layoutmode`. The first disables the emulation of Android callbacks (due to button clicks, GPS location changes, etc.), while the second ignores Android GUI components, like input fields, as dataflow sources.

Tuning the pathfinding precision by setting the `aplength` parameter resulted in six of the twelve remaining APKs succeeding. The `aplength` parameter accepts an integer and sets the FlowDroid *access path* length. Consider the Java statement `X.Y.Z = 5`. The left-hand side of the assignment statement has an access path length of 2. Using this parameter, one can configure how deeply FlowDroid will search access paths when tracking taint. With an access path length of 0, all the fields of an object referenced by a variable are conflated, potentially resulting in many false alarms.

Using the `nopaths` parameter was required for the remaining six APKs to make it through FlowDroid without timing out. The `nopaths` parameter causes FlowDroid to invest no effort in constructing accurate paths. Rather, it will simply report source/sink pairs.

Additional options to improve performance are described on the FlowDroid webpage.[1] For future testing, we will compare these options in terms of precision versus computation requirements, as we did for the work discussed in this report.

6.6 Conclusions and Future Work on Infrastructure Improvements

We substantially improved the success rate of Phase 1 analysis and investigated the performance of FlowDroid. We found in many cases that vertical scaling is not a viable solution to FlowDroid timeouts. Moreover, we found that some APKs will not yield to (timely) FlowDroid analysis without very liberal compromises on path precision.

There are two broad directions for future work in this area. First, more effort can be invested in Phase 1. FlowDroid could potentially be made to be more resource efficient. Also, more configuration options could be added to FlowDroid to allow more granular tuning of the pathfinding process. Further, the results reported here do not include the modifications made as part of the rest of the project (initial enhancements for analysis of dataflows that traverse shared static fields, Service components, and BroadcastReceiver components). These additions increase the work required by the Phase 1 analysis, in some cases non-trivially. At the time of this writing, preliminary results show increased timeouts when FlowDroid is run with these additions using the same hardware configuration. In summary, building a Phase 1 analysis that accurately finds a wide variety of flows, in a timely fashion, is a difficult problem — one that requires a variety of compromises.

Next, Phase 2 analysis needs some attention. There are some interesting questions yet to answer. How does tuning down the precision of Phase 1 affect Phase 2 results? Now that more APKs can be processed by Phase 1, can we find legitimate, inter-application flows in the wild with Phase 2 analysis?

[1] https://github.com/secure-software-engineering/soot-infoflow-android/wiki

7 Related Work

Felt et al. [6] found that about one-third of Android apps (of the 940 they tested) asked for more privileges than they actually used. They also found evidence that one cause of over-privilege is developer confusion, in part due to insufficient Android API documentation. Malicious apps can use permission re-delegation attack methods [7] that, when successful, take advantage of a higher-privilege app performing a privileged task for an application without permissions. The Epicc [3] tool precisely analyzes inter-app communication in Android. By looking at intents sent and intent filters (both declared in the manifest file and dynamically registered in the case of BroadcastReceivers), Epicc can detect vulnerabilities due to app communication, such as activity hijacking. Although it examines vulnerabilities due to intents, the Epicc analysis alone does not trace and identify data flows between sources and sinks. TaintDroid [8] does real-time taint tracking to dynamically detect data leaks, including flows that traverse multiple apps.

The most similar work to DidFail is IccTA [9],[1] which was developed at roughly the same time as DidFail. Like DidFail, IccTA statically analyzes app sets to detect flows of sensitive data. IccTA uses a single-phase approach that runs the full analysis monolithically, as opposed to DidFail's compositional two-phase approach with fast install-time analysis. IccTA is more precise than the current version of DidFail. Because of its greater context sensitivity, it does less overestimation of tainted data reaching sinks.

8 Conclusions and Future Work

Improvements to DidFail helped it succeed with more applications and detect more flows. These improvements focused on static fields, BroadcastReceiver components, Service components, and integration with new versions of Soot and FlowDroid. The new setup for instrumented cloud-based testing enables us to take advantage of Amazon's powerful virtual machines and to use virtual machines in parallel for faster test completion. We will continue to use the instrumentation in the new setup to monitor and optimize DidFail's performance. DidFail was modified to use new versions of FlowDroid and Soot, which include a better module for converting between Android DEX representation and Jimple intermediate representation. The new version of DidFail was able to successfully process three times as many apps as it was able to process previously.

The enhancements discussed in this report move us significantly closer to the goal of analyzing taint flows through all types of components and shared static fields. There are still more dataflows that need to be detected and analyzed, but our work has helped improve DidFail as a practical security tool. With a few more improvements, we hope that DidFail can ultimately be incorporated as part of a review process for vetting applications for malicious behavior.

[1] In addition to the arXiv preprint, a paper has been accepted at ICSE 2015 and will be published later this year.

References

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

[1] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android Taint Flow Analysis for App Sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis (SOAP)*, October 2014. [Online]. Available: http://doi.acm.org/10.1145/2614628.2614633

[2] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-Aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, January 2014. [Online]. Available: http://www.bodden.de/pubs/far+14flowdroid.pdf

[3] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. 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)*, Washington, D.C., 2013, pp. 543–558. [Online]. Available: http://siis.cse.psu.edu/epicc/papers/octeau-sec13.pdf

[4] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot — a Java Bytecode Optimization Framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1999, p. 13.

[5] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, January 1995, pp. 49–61.

[6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, October 2011.

[7] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *USENIX Security Symposium*, June 2011.

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

[9] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I Know What Leaked in Your Pocket: Uncovering Privacy Leaks on Android Apps with Static Taint Analysis," *arXiv preprint arXiv:1404.7431*, 2014.