Anaconda: Detecting private-information leaks in Android apps using static data-flow analysis

Rijksuniversiteit Groningen

Authors:
Stephan Groenewold
Klaas L. Winter
Jan Veldthuis

Supervisor:
dr. Doina Bucur
Second reviewer:
dr. Arnold Meijster

Abstract

With the advent of the smartphone, new ways of communicating and connecting with the internet have opened up. For many people, smartphones have replaced their watch, their address book, and their calendar. This means that the average smartphone has a lot of private information stored on it, for example: SMS or MMS messages, what web-pages were visited in the browser, or the call history. Unfortunately, it is not clear what apps on smartphones do with this information, and whether this information is used maliciously. Anaconda addresses these issues by using static data-flow analysis on Android apps, reporting whether these apps request private information, and reporting whether this information is sent to remote servers (i.e. leaked). In 14 popular apps, Anaconda found 572 requests of private information, of which 243 cases led to a possible leak. While some of these information leaks are legitimate uses, a large number of the leaks are not legitimate or at least suspicious. Most leaked information is either sent to ad servers or the app developers' servers, and may not even be used by the app itself.

Contents

1 Introduction **4**

2 Background **4**

3 Concept **6**

3.1 Why static analysis 6

3.1.1 Dynamic analysis 6

3.1.2 Checking outgoing data at runtime 8

3.1.3 Static analysis 8

3.2 Sources and Sinks 9

3.2.1 Direct sources and sinks 9

3.2.2 Indirect sources and sinks 9

3.3 Tracking from source to sink 10

3.3.1 Tracking forward from source 11

3.3.2 Tracking sinks forward 11

4 Realisation **12**

4.1 Finding sinks 12

4.2 Finding sources 12

4.3 Finding a path from source to sink 13

4.3.1 Actions 13

4.3.2 Difficulties 14

4.3.3 Optimisation 14

4.3.4 Track paths 15

4.4 HTML report 15

4.5 Algorithmic complexity 16

5 Further research **17**

5.1 Tracking references 17

5.2 Conditional statements 18

5.3 Usage of source to sink paths 19

6 Evaluation and results **19**

6.1 Analysed apps 20

6.1.1 Dropbox 20

6.1.2 Barcode Scanner (zxing) 20

6.1.3 Humble Bundle Downloader 20

6.1.4 Tweakers 20

6.1.5 Sudoku Free 21

6.1.6 Reddit Sync 21

6.1.7 Word Search 21

6.1.8 gReader 21

6.1.9 Shazam 22

6.1.10 Buienradar 22

6.1.11 Twitter 22

6.1.12 WhatsApp 23

6.1.13 iGroningen 23

6.1.14 NS Reisplanner Xtra 23

6.2 Resulting HTML 24

6.3 Discussion of results 24

6.4 Analysis time 24

7 Future work 27
7.1 Inheritance . . . 27
7.2 Permissions . . . 28
7.3 Reflection . . . 28
7.4 Intents . . . 28
7.5 Content providers . . . 29

8 Related work 29
8.1 Androguard . . . 29
8.2 TaintDroid . . . 29
8.3 SAAF . . . 29
8.4 Stowaway . . . 30

9 Conclusion 30

References 30

Appendix 32

A Leak type glossary 32

B List of recognised sinks 33

C List of recognised sources 34

D Leak occurence tables 38

1 Introduction

More and more people are starting to use smartphones. Smartphones are, for most people, ideal devices, since they offer the same functionality that before required multiple devices or items. Examples of the devices and items that are being replaced by smartphones include: calendars, address books, cameras, flashlights, watches, multi-media players, game devices, etcetera. The list of functionalities smartphones possess is only increasing with new technology and new apps. Because a large number of the tasks done by a smartphone are personal in nature, a big amount of personal data is stored on a smartphone. Furthermore, because smartphones are getting equipped with more and more sensors, such as GPS, motion sensors and a microphone, information from these sensors also becomes available. Examples of personal information that could be acquired because of this are: SMS and MMS messages, contacts, photos that were made, whereabouts of the phone, conversations that happen in close proximity of the phone, browser history, etcetera. Everyone that has access to the phone, including people and installed apps, could potentially access this information.

Although smartphones with different operating systems installed are available on the market today, this paper focuses on the Android OS. One important reason for this is the open source nature of Android, which makes working and developing with it easy. Another important reason is the fact that Android is by far the most popular mobile OS being sold today, with 74.4% of sold smartphones in the first quarter of 2013 using Android [1].

The Android OS tries to limit third-party apps' access to personal information by requiring specific permissions for different types of information, such as permission to connect to the internet or permission to access the user's address book. These permissions have to be given by the user upon installing a third-party app. However, because Android cannot notify the user what the app will actually do with that information, many users simply accept whatever permissions the app requests. Because the warnings produced by Android are often ignored, the system does not help much in securing users' data.

Not all apps are very careful with your information either. Take for example WhatsApp, the well known messaging service. In the year 2012, it has been found to not encrypt your messages, both when stored or send, allowing any other app with SD card access, or on the same network to be able to read your messages [2]. Numerous more security issues have been discovered since. Not only that, it sends all the phone numbers in your address book to the WhatsApp servers, to check which of them have been registered with WhatsApp. Who knows what happens with that information. Clearly, even widely used popular apps can not always be trusted with your personal information.

This paper presents Anaconda, a static dataflow analyser that is capable of detecting personal-information usage in apps, and is capable of determining whether this personal information is leaked by apps. Anaconda can be provided with an Android app installation package which it decompiles. Anaconda then searches for usage of personal information and tracks this information through the decompiled code. This tracking occurs in a depth-first-search fashion, and during the tracking a track tree is generated that shows what was done with the personal information, and most importantly whether the information is possibly leaked.

By providing information about whether apps acquire or even leak personal data, developers can make their apps more secure and privacy aware. It could also provide users with more detailed information about what the apps they install do with their personal information. This way they could look for alternative apps or use a solution such as TISSA [3] to not allow the leaking to happen.

In the next chapter we will first give a brief overview of the Android operating systems and some of the most important related terms we will use in this document. After that, we take a look at the theory behind some commonly used approaches to app analysis, and look into static analysis, and how Anaconda performs static analysis in more detail. A detailed description of each of the steps performed by Anaconda to analyse an app follows in the Realisation section. In Further research, we analyse some solutions to problems we encountered while developing Anaconda. Having discussed how Anaconda works, we look at the results generated by Anaconda. We discuss each of the 14 apps we analysed, in which Anaconda found a total of 572 requests of private information, of which 243 are leaked. The performance of Anaconda is also examined. Finally, we talk about some potential future improvements for Anaconda, compare our solution to what already has been done, and conclude our report with a conclusion.

The source code for Anaconda is available at `https://github.com/KPWhiver/Anaconda`.

2 Background

Android is an operating system, created by Android Inc., specially designed to be run on mobile devices,

Figure 1: Android Architecture

such as smartphones and tablets. In 2005, Android Inc. was bought by Google, who continued the development of Android as an open source project. Android uses a modified Linux kernel as its kernel. One of the features that were added to the Linux kernel by Google is the *binder driver*, which allows processes to communicate with each other. Some of the components under "Linux Kernel" in Figure 1 were also added by Google, such as more aggressive power management. Android supplies a set of C libraries, such as an implementation of OpenGL, a graphics library, and libc. Figure 1 shows a list of some available libraries. The libc provided by Android is based on the BSD C library and modified by Google. Android's libc is called Bionic and it is highly optimised for mobile devices.

Applications that are written for Android are, at least partially, written in Java. Android applications are commonly referred to as apps. To run this Java code efficiently, Google created the Dalvik Virtual Machine, which is a Java Virtual Machine optimised for mobile devices. The big difference between a normal Java VM and the Dalvik VM is that Dalvik works with register-based bytecode, while normal Java VMs work with stack-based bytecode. Both Dalvik and the standard Java libraries are part of the "Android Runtime" in Figure 1.

On top of these libraries and Dalvik, Android supplies a set of Java libraries for use by apps. This can be seen under "Application Framework" in Figure 1. These libraries include features like: a window manager, http clients, sensor access, etcetera. To obtain private information about the user of a smartphone, the supplied Java libraries can be used. When writing an app for Android, it is possible to use the provided Java libraries and the standard Java library, but it is also possible to use the C libraries that are available on the Android system.

For more information about the inner workings of the Android OS, see "A survey on Android vs. Linux" [4].

Several Android related terms will be used in this document. An explanation for what those terms mean can be found here.

APK: An APK (Application Package) file is an archive containing the code of the app in DEX format along with resources the app requires. These files are used to distribute Android apps. An APK is in reality just a zip archive, so the contents can easily be retrieved by using any unpacking tool.

DEX: DEX is a bytecode-format that is based on registers. Register-based bytecode means that if you look at a readable form of DEX you will see that all the data in a certain function is stored in certain registers, much like the way a typical Java program may store data inside local vari-

ables. DEX furthermore offers a list of different instruction types (opcodes) [5] to be used, such as method invokes, binary and unary operators, jumps, etcetera. A typical instruction in DEX is a certain opcode followed by a set of registers the opcode applies to. An instruction might also need other parameters such as how far an opcode should jump (in the case of a `goto` or an `if` statement) or what method to invoke (in the case of an opcode that invokes methods). Data about what classes are used and what methods they contain is still available in DEX.

Smali: Smali is a human readable form of DEX bytecode. The name originates from a DEX assembler with the same name, taking smali as input. The format of the data that Androguard [6] (see section 8) creates is very similar to smali, although a lot of information that is present in smali is not present in Androguard's format.

Activity: An Activity is an important component of any Android app and represents a single screen that has an interface which can be interacted with by the user. A single app can have many activities. For example, in a mail app, one activity might show a list of the emails in your inbox, and another might contain a form for sending an email. Each activity works independently, but can communicate with other activities, even ones outside of the current app, if that app allows it. All the started activities form a stack, with the one on top being the active and visible one. Activities in the background will be paused automatically, and can be stopped by Android to free up resources. Whenever an activity ends or the back button is pressed, the activity is popped off the stack and the previous activity is shown. If needed, it is restarted.

Service: A Service is also a component of an Android app. Like activities, multiple services can be contained in a single app. Unlike activites, they do not provide a user interface, and run in the background. Services are often used for long background tasks like downloading a file or playing music that should not stop when changing activities. Any app can communicate with a service (unless declared private). Examples of some system services are the notification service, volume service and alarm service.

Intent: Intents are messages that can be sent between Activities or to a Service. These messages contain an action the receiver needs to perform and data on which the action should be performed. An intent is either explicit, meaning the exact recipient is given, or implicit, when no recipient is specified and only criteria for the recipient is provided. Android uses this information to deliver the intent to the correct receiver, asking the user or picking one randomly when multiple are available. For example, an app might want to provide the user the ability to share something with other people. The app could then send an intent containing the action "ACTION_SEND". Android then allows the user to pick an app that provides this functionality, like your mail app or the facebook app.

Reflection: Reflection is a technique that allows a program to determine, at runtime, what kind of methods and fields classes have. In the case of Java it also allows determining what classes the program has, and it can be used to read and write to class members, including private class members. Reflection is posing difficulties in the context of static code analysis, because it allows a program to decide at runtime which function it will call. Static code analysis can only look at what is known at compile-time, making reflection hard to deal with.

Listener: A listener is an object intended to respond to certain events. For example, a developer can define a class with a method called `onMouseClick()`. After defining the class he can pass an object of the class to the class that handles the mouse, for example, a `Window` class. The `Window` class will store the passed object, and every time a mouse click happens it will call `onMouseClick()`. In this example the class with the `onMouseClick` method is the listener.

NDK: The NDK (Native Development Kit) is a way for developers on Android to call C/C++ code from Java. The C/C++ code that is called has access to the C libraries that are mentioned in the beginning of this section.

3 Concept

3.1 Why static analysis

There are several approaches to analyse apps for the purpose of information-leak detection. Let us look at the different approaches and see why we decided on using static code analysis.

3.1.1 Dynamic analysis

One way to check whether an app is leaking private data is by tracking private data at runtime through

the app. TaintDroid is a good example of this technique. As explained in Section 8, TaintDroid [7] modifies Android's Virtual Machine. Thanks to the modifications, the Virtual Machine is able to mark whether a piece of data in memory is tainted, and it is able to track this tainted data as it flows through the Android system. Because almost all of the apps on Android are actually being run on the Virtual Machine (the exception being apps completely run through the NDK), it even becomes possible to track data sent from one process to another process.

Code obfuscation

The big advantage of tracking taint dynamically is that when running the code you know exactly what code is being executed and what code is not, making *obfuscating* the code almost impossible. A writer of a malicious app may for example do the following:

Example 1: Calling a method through reflection

```
1  // encryptedMethodName is "getDeviceId"
2  // when decrypted
3  String encryptedMethodName =
4        "v84yvb4yt2rbc3rcnb832n08vb";
5
6  String methodName =
7        decrypt(encryptedMethodName);
8
9  //methodName now equals "getDeviceId"
10 Method deviceIdGetter = TelephonyManager.
11       class.getMethod(methodName);
12
13 String imei = deviceIdGetter.invoke();
```

In the example an encrypted version of the string "getDeviceId" is decrypted. After decrypting the string, it is used to make a call to the function with the same name through reflection. Because the function name is encrypted, if we just look at the code it is not clear what method is being called. If we were tracking dynamically, the Virtual Machine would see that the function that's actually being called here is `getDeviceId()`, which returns the phone's IMEI (an unique identifier associated with the phone), because that's the function the Virtual Machine looks up through reflection. Using static analysis it is almost, if not entirely, impossible to figure out which method is being called here.

Control flow

The fact that dynamic code analysis only looks at the code being executed is both a strength and a weakness. Why this is, will be illustrated in the next two examples concerning control flow. Example 2 shows a strength of dynamic analysis.

Example 2: Leaking based on a boolean

```
1  int taintedData;
2  // someBooleanValue is set to false somewhere,
3  // because the code should never run
4  if(someBooleanValue)
5    leak(taintedData);
```

In this example we only leak if `someBooleanValue` is set to true. Because `someBooleanValue` is always false, the Virtual Machine will never get to the leak code and no leak will be reported. With static code analysis, `someBooleanValue` must be tracked to figure out whether the boolean variable could possibly be true. Unfortunately, it can not always be determined whether a boolean variable can actually become true. The topic of tracking conditional variables is further discussed in Section 5.2. An example of a leak that occurs based on some boolean value can be found in the Amazon ad-API, where location data is only sent to Amazon if the developer who is using the API calls `enableGeoLocation(boolean)` in the `AdTargetingOptions` class, with the boolean being true. In this case it is very easy to see that, although the developer never intents to leak information, static code analysis might still report a leak because code that is able to leak is present.

In the following example, a weakness of dynamic code analysis can be seen:

Example 3: Leaking only if possible

```
1  Location GPSloc = getLastKnownLocation();
2
3  if(GPSloc != null)
4    leak(GPSloc);
```

In this example we only leak data if requesting the data did not fail. Dynamic code analysis will only see the leakage if the request does not fail. If the request does fail, the leak code will never be executed and dynamic code analysis will never come across it. Static code analysis can detect the leakage because it will just assume the worst case, in this case that `GPSloc` can be unequal to `null`.

Other disadvantages

In the case of TaintDroid (although this also goes for dynamic analysis in general), it is very hard to dynamically track data through a C/C++ program, if not impossible. This is because unlike

Java, C/C++ code is not run inside a Virtual Machine, instead it runs directly on the hardware. This means that TaintDroid can only make assumptions about what happens to private information after it is passed to C/C++ code.

Other disadvantages of dynamic code analysis are that it introduces CPU and memory overhead at app runtime. Also, because you generally do not follow all possible execution paths when running an app, you could potentially miss certain leakages. Example 3 is a simple example of this behaviour. Because the execution path usually depends heavily on runtime conditions, it becomes very likely that we do not follow all execution paths.

3.1.2 Checking outgoing data at runtime

Another runtime technique to detect information leakage is by sniffing the data packages that are actually leaving the phone. This could mean that you filter the data that is being sent, to see if, for example, the phone's IMEI is present in the output. Although this system will probably not give a lot of false positives, it is very limited in what it can detect.

Some data, like data that shows where you are, e.g., longitude and latitude, will constantly change, and it will therefore be very difficult to see whether something is an irrelevant piece of data or an actual location coordinate. One way to get around this is to modify the Android-API in such a way that it feeds the app false data, for example, false location data. By feeding the app false location data, we can match against the false data to see if the app leaks it. The TISSA system, is a system that does exactly this: it provides apps with bogus private information so no real private information is leaked. A downside to this is that providing bogus private information might render some apps useless, e.g., an app which tells an user his or her location needs access to location data.

Unfortunately this technique fails completely when the data that is sent out is encrypted, which is becoming more and more commonplace in software, for example through techniques such as SSL and HTTPS.

3.1.3 Static analysis

A third way, and the way we chose, to detect private information leakages is by using static code analysis. Static code analysis is a way of analysing the behaviour of a program without running the program. To achieve this, some analysable version of the program needs to be available, for example, the source code or the assembly version of the program. Static code analysis is used a lot in detecting bugs in source code [8], but it is sometimes also used to detect malicious behaviour in programs, as is the case with some of the projects in Section 8 (Related Work).

There are a number of benefits static code analysis has that other analysis techniques may not have. Also, since any analysis that looks at code, without actually running the code, falls under static code analysis, there are many ways in which static code analysis can be done.

Benefits of static code analysis

Because static code analysis is not restricted in what code it can look at, all the code of a program can be analysed. This means that issues that would only occur on, for example, a different CPU architecture can still be detected. Even code that will never be executed, e.g., a function that is never called, can be analysed. Example 3 is a good example of how static code analysis can detect issues in code that is not executed.

Because static code analysis can analyse all the code at once, it only needs to be run once. Depending on what kind of analysis is being done running the analysis might still take long, but it will have no effect on the runtime performance of the analysed app.

Using static code analysis, it is possible to build an analyser which reports no false negatives, e.g., it is possible to have a static analyser that does not miss any leaks. This is mostly due to the fact that static code analysis offers the ability to look at all the code at once, not just at the code being run. As such all leaks can potentially be spotted, not just leaks in the code that is actually being run. Even when reporting no false negatives, it is still, almost always, possible to give detailed information about reported leaks, for example, in the form of an execution path which causes a leak. The only cases in which no detailed information can be given are the cases in which it is only possible to assume that leaking occurs, e.g., when information about the leaking is only available at runtime, as is the case with reflection. A downside to giving no false negatives is that static code analysers usually give, at least some, false positives. Chances are big that when an analyser is made to give fewer false negatives, more false positives are generated.

Static code analysis methods

Different methods to do static code analysis usually range from giving in-depth leak reports to just giving an superficial overview of program behaviour. This can clearly be seen in the previous attempts of statically analysing Android apps.

Tools like Androwarn [9] and Andrubis [10] only look at whether private information is being requested in a program. The upside to doing this is that it only requires a lightweight analysis. The downside to this method is that the results are not very extensive. SAAF [11] goes a step further because it first applies program slicing to the code, to determine whether the code to request private information can be reached and is not simply "dead code". Program slicing is a technique that "slices" all the code away that is not relevant to the concrete problem currently being looked at, in this case code that can not be executed. In the end, these three tools only give an overview of what private information is requested.

Anaconda gives more information about what an app does with private information, because it also determines whether requested private information leaves the phone. Anaconda achieves this by also tracking the private information through the code of the app. The results Anaconda produces are much more precise, but it takes more effort and time to get these results.

For even more precision, it may be possible to use model checking to analyse an app. Model checking creates a mathematical model of the program code. After creating the mathematical model, a check is performed to see whether the model satisfies a given formula. The given formula would in this case be a formula that describes malicious behaviour. Model checking can be more precise than simpler forms of static code analysis, such as the analysis Anaconda performs, but it is also much more expensive in terms of the CPU time and memory usage it needs to perform its analysis [12].

Why static code analysis
In the end, we chose to use static code analysis to attack the problem of leaks in Android apps. Even though static code analysis is limited in what it can analyse [13], static code analysis has several benefits over other techniques that allows it to detect leaks other techniques can not detect. Furthermore, dynamic code analysis has already extensively been looked at with taintdroid, while existing static code analysis solutions are still severely lacking (as can be seen in Related Work, Section 8). All in all we felt that the best results and the best progress could be achieved by using static code analysis.

3.2 Sources and Sinks

To be able to detect whether an app leaks information, we need to know a few things. First, we need to know whether an app is actually acquiring private information, and where it takes place. Second, we need to know where the places are that the information could leave the system and therefore be leaked. To approach this problem, we introduce the terms *source* and *sink*, where a source is the program location that provides private information, and a sink is a program location from where data can be sent to the internet.

Basically there are two types of sources and sinks that we have to deal with: *direct sources* and *direct sinks*, and *indirect sources* and *indirect sinks*. Direct sources are sources that we can be sure of that they provide private information, while direct sinks are sinks that we can be sure of that they will send the provided data to the internet. Indirect sources and sinks are ways in which information can leave or enter an app to or from the rest of the operating system, for example, file reads and writes.

3.2.1 Direct sources and sinks

The only direct sources in the Android system are certain methods and structures in the Android-API. An example of this is the `TelephonyManager.getDeviceId()` call, which returns the phone's IMEI number. The Android-API has quite a lot of direct sources, both in the form of methods that directly return private information, but also in the form of methods that can be passed listener objects. The passed listener objects will eventually be passed private information.

A direct sink in the Android system might be, for example, a socket or an http-request.

3.2.2 Indirect sources and sinks

Example 4 shows an example of indirect source and sink usage in an app:

Example 4: Leaking through a file

```
1  function1:
2      int privateData;
3      writeToFile("file.txt", privateData);
4  function2:
5      int data;
6      data = readFromFile("file.txt");
7      leak(data);
```

When we are reading data from outside the app, such as in `function2`, we do not know where this data is coming from. The data we are reading might come from a direct source, such as in `function1`. We can not be sure that data that is written outside the app will not, later on, be read and leaked. Because of this, we have to deal with these potential sources and sinks, which we call indirect sources and sinks.

There are numerous ways in which an app can make data "leave" the app and, later on, "enter" it again:

- **Files:** File writes and reads, such as in Example 4.
- **NDK:** By using the NDK a developer could pass data to a C/C++ method and later request that data again. Since this is native code it becomes very hard to track this code statically. It is also possible for native code to read and write data to and from files and to use sockets.
- **Intents:** An app could send or receive an intent containing private information, that could later on be leaked.
- **Reflection:** Although reflection is not really a way for getting data outside of a running app, it is still useful to treat reflection as an indirect source or sink. Every function call done through reflection could potentially return private information or leak private information.

There are several ways to deal with indirect sources and sinks, ranging from algorithmically simple producing a large number of false positives to algorithmically complex producing almost no false positives.

A. **Track whether an indirect sink is accessed again**
One way to deal with this problem is to track whether anything we put in an indirect sink is accessed again. For example, if we write something to a file, search if we also read from that file, if we do read from that file we can continue tracking the data that is read from that file. The downside to this method is that it can be hard to find whether we access an indirect sink again. We would have to find out whether, for example, a file that is read from is exactly the same as the file we wrote to. Especially file names can be easily obfuscated. For some types of indirect sources and sinks such as the NDK this method is close to impossible to implement without looking at the C/C++ code that is used. Statically analysing C/C++ code is certainly possible, but it is not something we have looked into.

   In Figure 2, the apps in set 4 correspond to the apps reported as leaking with this solution.

B. **Assume all indirect sources of the same type, are sources**
Another way to deal with this problem would be to simply state that whenever we write something to a file or another indirect sink, all reads of the same type return tainted information. For files this would mean that whenever private information is written to a file, all file reads are assumed to return private information. This way of dealing with the problem is much simpler, because we do not need to be sure that we are dealing with the same file or intent. The downside of this approach is that it can result in a lot more false positives.

   In Figure 2, the apps in the intersection of set 2 and set 3, correspond to the apps reported as leaking with this solution.

C. **Make all indirect sinks direct**
By far the easiest solution is to simply state that if something is leaked through an indirect sink it is leaked, and when something is accessed from an indirect source it is private information. The big downside of this is obviously the huge amount of false positives it can create. It must be noted that in the case of reflection, it is also necessary to treat the indirect sources as direct sources. The reason that indirect sources need to be treated as direct for reflection is because a call using reflection which returns data (indirect source) can potentially be a direct source, an example of this is Example 1.

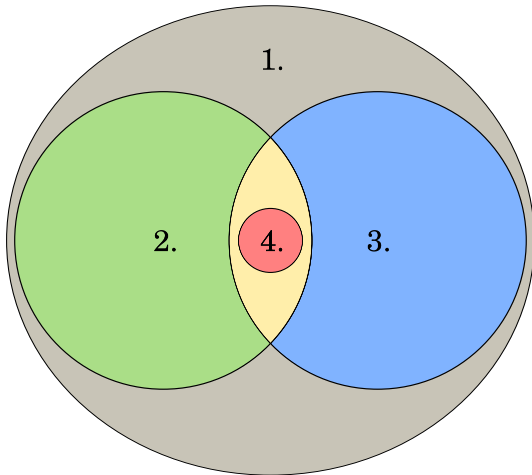
   In Figure 2, the apps in set 2 correspond to the apps reported as leaking with this solution. In the case of reflection, the apps reported as leaking will be a union of the apps in set 2 and set 3.

Anaconda uses solution C to solve the problem of indirect sources and sinks, with the exception that reflection is currently not dealt with. Changing the solution used to solution B or A is something for potential future work.

3.3 Tracking from source to sink

To actually make the connection between acquiring data from a source and passing that data to a sink, we need to track what happens to the data that is provided by sources. This whole process not only involves tracking the data from source, but it also involves tracking sinks and tracking references to the data. The subject of tracking references is discussed in Section 5.1.

Figure 2: Venn diagram



1: All apps.
2: Apps that leak private data to an indirect sink.
3: Apps that leak data acquired from an indirect source.
4: Apps that leak private data through an indirect source and sink pair.

3.3.1 Tracking forward from source

The first step in figuring out whether private information is leaked is by finding out if and where in the code this information is acquired. Finding where private information is accessed can be as simple as figuring out where certain functions, such as `getDeviceId()`, are called. After finding where this data is accessed, we can track the data through the app, to find out if it eventually leaves the phone. To track data we simply look at all the instructions that use the data, and take the appropriate action. Let us look at an example of an attempt to leak the phone's IMEI number.

Example 5: Simple leak

```
1  String imei = manager.getDeviceId();
2
3  Socket socket = new Socket("hostname", port);
4  DataOutputStream out =
5      new DataOutputStream(
6      socket.getOutputStream());
7
8  out.write(imei);
```

In the example, `getDeviceId()` is called. Since `getDeviceId()` returns the phone's IMEI, we consider it to be a source and track the result. The result of the function call is stored in the local variable `imei`, so we start tracking `imei`. Eventually `imei` is passed to an output stream connected to a socket, so we can now report a leak. In this example the instructions that we track are simple: an instruction that stores the result of `getDeviceId()`, and an instruction that invokes the method `DataOutputStream.write(String)`. In more complex examples there might be instructions that return the data, data might be stored inside a class member, etcetera.

3.3.2 Tracking sinks forward

One problem that appears in the previous example is the fact that, while tracking, we do not really see the link between the `DataOutputStream` that we write to, and the `Socket` that this `DataOutputStream` eventually sends its data to. To still be able to detect whether data will actually end up in, in this example, a socket, we need to identify which instructions cause data to be passed to a sink (such as the invoke of the `DataOutputStream.write(String)` method).

Identifying which instructions pass data to a sink can be done by tracking from the place where the sink is defined. Finding where a sink is defined can, for example, be done by looking at where the constructor of the sink is called. After having found the sink, we look for instructions that call a method of the sink. The only thing left to do after we have found such a method is marking this instruction as passing data to a sink. Furthermore, we also need to handle the other instructions that try to do something with the sink we are tracking, for example, when the function we are in returns the sink. Luckily we can handle this by using the same rules we used when tracking forward from source.

In Example 5 we treat the `OutputStream` that `Socket.getOutputStream` returns as the sink, since this is the object that we will pass data to. While tracking this `OutputStream` we would notice that it is passed to a `DataOutputStream`.

Because of this, we also start tracking the `DataOutputStream`. Eventually a method call of a method of the sink we are tracking, the `DataOutputStream` is called, meaning we can mark this instruction as passing data to a sink. When the tracking from source occurs we will find that `imei` is passed to the method we marked as passing data to a sink (`DataOutputStream.write(String)`), and a leak can be reported.

In the end, we only need one algorithm for both tracking data forward from a source and tracking sinks forward. The main exception this algorithm makes is that while tracking sinks, instructions may be marked as passing data to a sink, which is not the case when tracking from source.

4 Realisation

To determine if an app leaks, Anaconda [14] tries to find a path between a source and a sink. To be able to find said sources and sinks we first need to convert the APK, and more specifically the DEX inside, into a format that we can use to analyse the app. Instead of doing this ourselves from scratch, we used the Androguard tool. As explained in Section 8.1, Androguard is a collection of tools for analysing APKs. Besides decompiling the DEX, it provides simple functionalities for looking up usages of fields and function calls. We use this to find the locations of usage of certain functions or fields we are interested in. We then further analyse the instructions at these locations to track the data through the code. The following four steps can roughly be distinguished in this tracking process in Anaconda:

1. Finding sinks
2. Finding sources
3. Finding a path from source to sink
4. Generate HTML report

We will take a more detailed look at each step below.

4.1 Finding sinks

The goal of this step is to find all the direct and indirect sinks created in the app and mark the instructions where they are used. When tracking private data from sources to a sink in the third step, finding a path from a source to sink, we can use this information to determine if the information is leaking or not. First we start looking up all the locations where a sink is created. For this we use Androguard. We created a list of sinks available on the Android platform, containing 49 of the most common sinks. In this list are, for example, the standard Java socket but also the Near Field Communication (NFC) adapter. We pass each entry to Androguard, which then tells us where the sink is used. The entire list can be viewed in appendix B.

After having found the used sinks, we track these sinks to find all leaking instructions as described in tracking sinks forward in Section 3.3.2. While tracking we take actions depending on the instructions encountered as described in Table 1. After completing this step, each instruction involving a function call on a sink has been marked.

It is important to note that in the case where a function is called that is not defined in the APK, we only mark it as leaking if the call is made on the sink object, not if it is used as a parameter. What this means is that we do not take cases where a sink is passed as a parameter to a standard library call in consideration. We could not find any cases where this would result in an actual leak, and looking manually through cases found by Anaconda all cases appear to be false positives. As such, including these cases would only result in more false positives and we decided to exclude them. When, however, the code is available, we continue tracking that parameter in the function itself.

4.2 Finding sources

Now that we know where all the sinks are, we need to check if any private information from sources is put into them. To find the sources, we again have constructed a list of sources we are interested in. This includes functions like **getDeviceId()**, which returns an unique device id, and **getLastKnownLocation()**. Calling a function, however, is not the only way of accessing private information. Private information could also be stored in fields of objects. An `Account` object, for instance, has a `name` field. No function call is needed to retrieve this information. So, besides function calls, this list also includes fields that contain private information. Another way to get private information is adding a listener. A `LocationListener` could be used to retrieve the current location of the user. Each time the location is updated this Listener is notified and passed the new location. Some known listeners are also included in this list. Combining these methods, fields and listeners results in a list of 115 known sources, which is attached in appendix C.

Just like with the sinks, we feed this list into Androguard which tells us where the sources are used.

Table 1: Decision logic. ***X*** represents the register currently being tracked, where ***X*** is either a sink or a source. Other registers are referenced by ***v***. Constants are represented by ***c***.

Instruction	Action	Description
move X, v	*Stop*	Tracked register is overwritten
move v, X	*Track(v)*	Information is copied into v, track v as well
return X	*Track(Method.usage)*	Continue tracking at every call of this method
const X, c	*Stop*	Tracked register is overwritten
invoke {X, ...} Method	*Track(Method.return)*	First parameter is the instance, so no point tracking in method. Track what is returned
invoke[1] *{..., X, ...} Method*	*Track(Method.parameter(X))*	Continue tracking in the method
invoke-virtual {..., X, ...} Method	*Track(Method.parameter(X))*	Continue tracking in the method, and in this method of subclasses as well.
invoke[2] *{..., X, ...} Method*	*Track(Method.return)*	Method is not defined in APK, track what is returned
invoke-static {..., X, ...} Method	*Track(Method.parameter(X))*	Continue tracking in the method
invoke-static[2] *{..., X, ...} Method*	*Track(Method.return)*	Method is not defined in APK, track what is returned
check-cast X	*Pass*	Tracked register is not modified
new-instance X	*Stop*	Tracked register is overwritten
iget X, v, Field	*Stop*	Tracked register is overwritten
iget v, X, Field	*Track(v)*	Field of tracked object is accessed
iput X, v, Field	*Track(Field.usage)*	Continue tracking at every *iget* of this field of any instance of this class
iput v, X, Field	*Pass*	Field of the tracked object is changed
sget X, Field	*Stop*	Tracked register is overwritten
sput X, Field	*Track(Field.usage)*	Continue tracking at every sget of this field
unary-operator X, v	*Track(v)*	Start tracking the result
unary-operator v, X	*Stop*	Tracked register is overwritten
binary-operator X, v_1, v_2	*Stop*	Tracked register is overwritten
binary-operator v_1, X, v_2	*Track(v_1)*	Start tracking the result
binary-operator v_1, v_2, X	*Track(v_1)*	Start tracking the result

A [1] above an invoke instruction means, in case of a sink being tracked, this instruction will be marked as passing data to a sink. A [2] means the method is not defined in APK, and we can not continue tracking in the instructions of the method.

4.3 Finding a path from source to sink

When we have found a source, we need to determine what happens with the private information retrieved from it. As explained before, the Dalvik Virtual Machine uses registers. By determining what register the information is put into, we can look for usages of the information by looking for usages of that register. Whenever the register is used, we first need to check if we previously marked the instruction as passing data to a sink. If this is the case, that means the private information is potentially being leaked! The instruction is marked as leaking and we continue with the next instruction, as it might leak in more locations. If it is not marked, we need to analyse this instruction to determine what happens with the information. Possibly we need to start tracking other registers that now also contain the private information, or at least parts or references to it. There are however a lot of different instructions, which all perform different operations on the register. Depending on the instruction, a different action should be taken. We describe these actions in Table 1.

4.3.1 Actions

There are three possible actions when an instruction uses a tracked register:

- In the first case, we need to track an additional register. This could be when, for example, a move instruction is performed. These kind of instructions move the contents of one register into another. Another example is a function call which takes private information as a parameter. It could return data which is based on the private information, which then needs to be tracked. Tracking a new register does, however, not necessarily happen in the same function we are currently tracking. Tracking a new register could very well happen in a called function, with the currently tracked register as a parameter.

- The second possibility is that we stop tracking a register. Let us consider the *move* instruction example again. What if the tracked register is not the source register, but the target register? That would mean the register that is being tracked is overwritten with new

information. This reference or copy of the information we were tracking is lost, so we no longer track this register.

- The final possibility is the most trivial one. The register is used in an instruction, but this does not result in the private information being transferred to another register. An example of this is the *check-cast* instruction. It checks if the object referenced in the register can be cast to a certain type. If this test fails an exception is thrown. The flow is possibly interrupted here, but the information in the register is not transferred or modified in any way.

These three actions can be represented by three different functions. These are *Track(x)*, *Stop* and *Pass* respectively. We have used these three functions to present our decision logic in Table 1. In this table, X represents the register currently being tracked. By $v, v_1, v_2...$ we denote any other registers used in the app. *Method* and *Field* represent references to a specific method and field. Most of these actions are quite self-explanatory. For example, when the tracked register is overwritten, tracking of it is stopped. When it is moved into another register (modified or not), the new register is tracked as well.

There are many more instructions available in the Dalvik Virtual Machine, but most are variations on the same instructions that we present in Table 1. In most cases the action taken does not change and as a result we omitted these instructions from the table.

4.3.2 Difficulties

Although the mentioned cases could be handled without much complexity, there are more complex cases. Most important is the differentiation between an *invoke* where the instructions are available, and an *invoke* where the instructions are not available. It is possible for a function to be called which is not defined within the APK we are currently analysing. While included libraries are enclosed in DEX, standard Java or Android classes and methods are not. As a result, we have no access to the instructions of that method and we can not dynamically find out what the function does. The only solution is specifying ourselves what a method will do with its parameters. We have done this for some sinks and sources, but it is hard to do this for every function available. In these cases, we have to make an assumption. When a tracked register is passed into a function we do not have the code of (and which is not marked as leaking data to a sink), we assume the tracked register is contained in the returned data in some form. While this clearly can result in false positives, it appears to be the only sound option in this case.

Another difficult case is instance fields. Instance fields are variables declared in each instance of a certain class. Each instance field is independent and can be different for each instance. Class fields are simpler. These are variables at the class level, and as such there is only one. When private information is put in a class field, we can look-up all locations where this field is read again. At every location we then continue tracking the register it was copied in. Instance fields, however, are harder to deal with. It is very hard to know which instance is used when an instance field is accessed while only tracking forward, as shown later in Section 5.1. To still be able to detect whether or not a field is used again, we look for all reads of the field instead. This means we potentially start tracking fields from other instances of this class. As a result, false positives will be introduced.

Detecting where a function is used or a field is read is relatively simple. But we defined a third kind of source, listeners. Listeners are harder to track down because they always are a subclass of an interface or abstract class. Looking for objects passed to **AddListeners** will often not be enough either, as at compile time these can very well be of the interface type instead of the subclassed type. To detect these listeners, we therefore look for any subclasses of the listeners. This is possible because listeners always need to be implemented by the developer. This does not yet guarantee it is also added as a listener, but when a listener is defined, it is likely to be used as well. We know which function of that listener will be called, and which parameter contains the private information. From here on we can treat it as a normal function call with a tainted parameter.

4.3.3 Optimisation

Because of the way tracking works in Anaconda, it is very likely that we start tracking paths that already have been tracked. Tracking paths that have already been tracked is very detrimental for performance, and should be avoided. There are two ways in which it is possible to start tracking paths that already have been tracked: loops in the code, and calling functions that were called before. The methods we have used to optimise these cases are very similar.

Loops
Code is bound to have loops in it, else the function-

ality of an app would be severely limited. Loops could appear in the form of a simple for or while loop, but also in recursion or goto constructs. To prevent Anaconda from going into an infinite loop while tracking, we need to check whether we are starting to loop. We have solved this problem by keeping track of every instruction we visited, and also remembering the register we were tracking when we visited it. These combinations of instructions and registers are remembered for as long as we are tracking the current function. Then, for each instruction we visit, we check if we have been there before with the register we are currently tracking. If we arrive at an instruction we have visited before, we appear to be in a loop. When we have detected this loop, we stop tracking in the current tracked path, which is equivalent to the *Stop* action in Table 1. Since the way we track is deterministic, when we encounter an instruction/register pair we have encountered before, we can be sure that we will, from there, follow an already followed path. We also do not have to be afraid that, when we encounter a loop, the register we are tracking contains something else than the first time, since we are still tracking the result of the same source.

Function calls

Whenever a function is called that has been called before, code that has already been tracked could be tracked again. Tracking code that has been tracked before, because of function calls, can also cause Anaconda to go into an infinite loop. An infinite loop, in this case, happens when, for example, function `A()` calls function `B()`, which in turn calls function `A()`. Not only can function calls cause infinite loops, they can also cause very long tracked paths to be tracked again, sometimes tens to hundreds of times, making some analyses take several minutes instead of several seconds. To solve this problem, we also keep track of the instruction and register we look at when we first start to analyse a function. These instruction/register pairs are stored in a global structure. Every time we start tracking in a function, we first check this global structure to see whether we have started tracking in this function before. If we have been in this function before, we stop tracking the current path, which is, just like with loops, equivalent to the *Stop* action in Table 1.

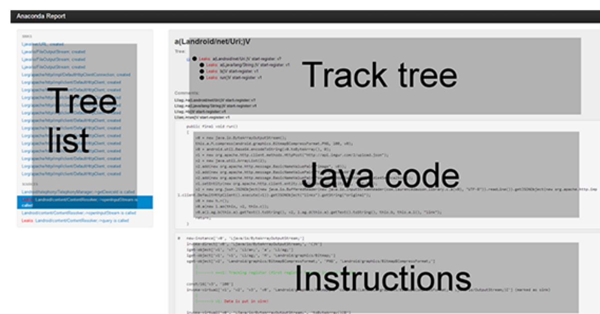
As opposed to the loops solution, it is possible for a register we are tracking to contain something different than the first time we visited this instruction/register pair. This could pose a problem when, for example, the code has a function `leak()` that leaks data. The first time data is passed to this function, e.g., the phone's IMEI, a leak of the IMEI is reported. The second time data is passed to this function, e.g. location data, no leak is reported, because we have been here before and thus execute the *Stop* action. Because we *Stop* the second time, we never report that location data is being leaked as well. To fix this problem, we need to know whether tracking from a certain instruction/register pair results in a leak. For this reason, this information is also stored in the global structure. When we find a leak, we mark every instruction we inspected in the path from the root to this leaking instruction as leaking. Now, when we encounter an instruction/register pair we have already tracked, all that has to be done is check if this instruction/register pair leads to a leak. If this instruction/register pair leads to a leak, we can also report the private information we are currently tracking to be leaking.

4.3.4 Track paths

Figure 3 shows an example of how different paths of tracking form a tree. The root of the tree is formed by the first instruction analysed. Each column represents a track path, and each row shows the actions taken for a single instruction. A single path only looks at a single register. If that register is overwritten, that path ends. Whenever the information is copied and/or modified, and then put in a different register, a new path is created in which tracking continues. This could be compared with depth first search. Each time a new register needs to be tracked, that register is tracked first. When that path ends, either because it is overwritten or the method ends, the algorithm continues on the previous path.

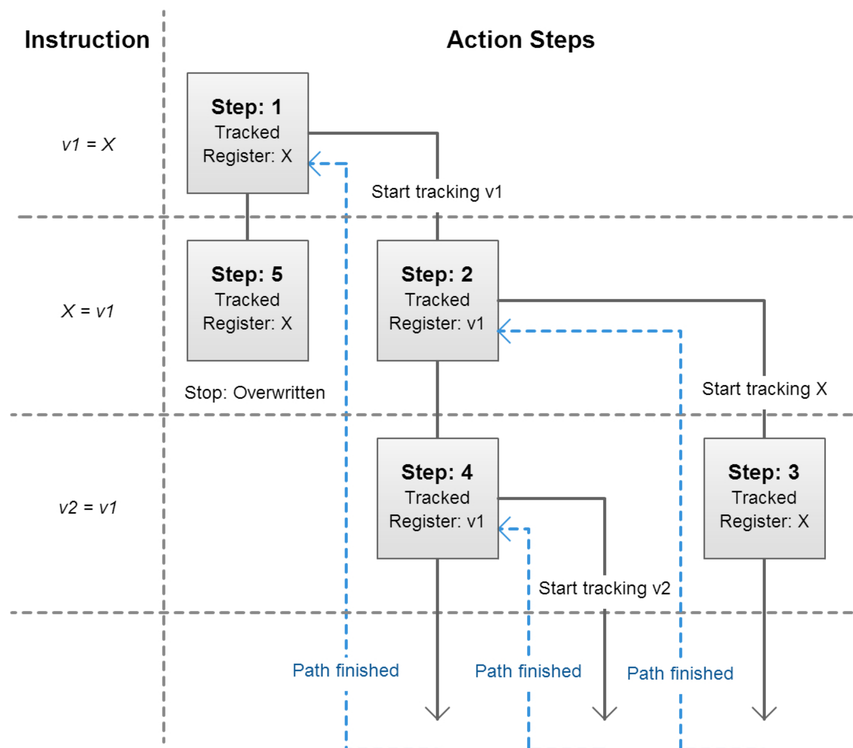
4.4 HTML report

Figure 4: HTML report



To help us get insights into the results of Anaconda, we generate an HTML report from the results. To do this, we build a tree structure like in Figure 3 for each source and sink from our lists found by Androguard, resulting in a list of trees. We keep track

Figure 3: Example track tree



of all the decisions made and all the instructions which have been marked as leaking and use private information, i.e., instructions that leak. From this information we build a report in HTML. An overview of such a report is shown in Figure 4. The report consists of four sections. On the left side is a list of all the different track trees for all the sinks and sources. When they contain a leak, they are marked as leaking. Selecting a tree allows you to view more detailed information about that tree on the right side of the page. At the top the tree itself is shown. This tree is, however, a simplified version of the tree shown in Figure 3. Instead of showing each path which tracks only one register, which would result in a lot of branches, we represent a single method as a node. Each node in the tree is marked as leaking if either it or one (or more) of its children leak private information.

Below the tree is the code of the currently selected node. Androguard provides functionality to generate Java code from DEX. As Java code is much more compact it is easier to read, but many names are lost and sometimes even obfuscated when compiled into DEX. It is only an approximation of the original code. Under the Java code are the separate instructions extracted from DEX. They are grouped into code blocks so that the progam flow becomes visible. The GOTOs indicate which block follows. Some of the instruction have comments, indicating where and which instructions are tracked, and which actions are taken. Instructions that are found to be leaking are marked as well.

While these paths can be quite lengthy, they do give good insight into where the private information comes from and which sink or sinks it is leaked into, including the entire path between them. As the instructions and actions are provided, it also allows for manual verification.

4.5 Algorithmic complexity

The time it takes for Anaconda to do its analysis heavily depends on the code of the app it is analysing. If the app that is being analysed does not use a big number of sources or sinks not much tracking needs to be done, as such the time needed for analysis is short. The time needed to analyse can also greatly differ between apps, based on how deep we need to track. Sometimes a track is stopped early, because the data that is being tracked is, for example, overwritten.

Even though analysis time depends on the app code, it is possible to give an upper limit to the amount of instructions we need to look at to analyse an app. Thanks to the earlier mentioned optimisations, an instruction will not be looked at

more than twice, once for finding sinks, and once for finding sources. The time it takes to handle an instruction depends on the number of arguments the instruction takes. For each argument an instruction takes, a new track could possibly be started. The number of arguments is limited by the number of arguments a function in this app can have. As such the upper complexity limit is $\mathcal{O}(i * a)$, where i equals the total number of instructions used, and a equals the number of arguments functions in this app can have. The lower complexity limit is $\mathcal{O}(1)$, this is the case where no sources or sinks can be found. Please note though that it is not known to us what the complexity is of the decompiling process of Androguard. Because of this the lower complexity limit, and even the upper complexity limit, could be higher.

5 Further research

Unfortunately time is limited. As a result we were not able to implement everything we would have liked to. Below we look into a number of problems in our current program. We did research solutions for these problems, and formulated the results into concrete solutions. Unfortunately, we did not have time to implement any of these solutions into our system. Still, we would like to share our proposals to resolve these problems.

5.1 Tracking references

Tracking data forward, as is done with private information and sinks, catches a lot of leakages that appear in code. There is, unfortunately, a type of leakage that we can not track by tracking forward from a source. The problem that arises when using merely forward tracking from a source can be easily shown with an example:

Example 6: Leaking through class fields

```
1  String imei = this.imeiMember;
2  imei.append(manager.getDeviceId());
3
4  // Leak something to the internet
5  leak(this.imeiMember);
```

Because Java is based on references, the previous described methods of tracking forward does not detect the leak that occurs. When `this.imeiMember` is assigned to the local string `imei`, nothing is copied, instead `imei` now points to the same data `this.imeiMember` is pointing to. Because `imei` is a reference, when we append to it we also append to `this.imeiMember`. When the actual tracking occurs, we start with tracking the result of `getDeviceId()`, which is appended to `imei`. Because of the append to `imei` we start tracking `imei`, however, `imei` is not used after the append, and no further tracking will be done. Eventually the code leaks `this.imeiMember`, which contains the IMEI because it references the same data as `imei` did.

Basically, the problem occurs because we are not sure who else possesses a reference to the data we are tracking. In this example someone else, `this.imeiMember`, possessed that data. To solve this problem we propose a number of solutions.

Tracking all class fields

One solution would be to track the field usages of all the fields in the app. By tracking all the fields it is possible to create lists of places where others refer the same data as the field we are tracking. Creating these lists would be done by looking where exactly a field is accessed, and after that tracking this reference forward and noting where it is used or copied.

In Example 6 we would first look where `this.imeiMember` is accessed. After finding the usage of `this.imeiMember` in our example, we can start tracking `this.imeiMember` forward. Since `this.imeiMember` is copied to `imei`, we start tracking `imei`. At all the occurrences of `imei`, we make a note stating that it refers to the same data as `this.imeiMember`. Once we have done this, and once we have determined that `imei` contains private information, we can also decide to track the usages of `this.imeiMember`, since we know that it refers to the same data as `imei`. A potential problem with this solution is that it requires that we track all the field accesses in the app. Since the number of field accesses is generally quite high, this could potentially have a big negative impact on performance.

Tracking references back

Another solution is to track back when we encounter the situation that private information is appended to a local variable. Appended in this context does not mean a literal append, as is the case in the example, it means the case where data is added to a variable instead of overwriting it. The idea here is to track back and see how this local variable has been created. It could be the case that this local variable has been created by a `new`, in which case nothing special has to be done. The interesting case would be when the local variable is created by an assignment from some other variable. When this other variable is a class member, we can

look for usages of this class member. If the other variable is not a class member we have to keep on tracking both variables.

In the previous example we would see that private information is appended to a local variable, `imei`, and as such we track back to see what happens to `imei`. While tracking back we will notice that `this.imeiMember` is assigned to `imei` and decide to track usages of `this.imeiMember` forward, finding the leak. This approach will take extra performance, but since we do not have to track all the fields, as in the previous solution, this approach is a lot cheaper than the previous approach.

5.2 Conditional statements

A problem that can occur is that whether something is actually leaked can depend on conditional statements. A common example of this behaviour is that APIs that provide ads for Android apps sometimes only request certain private information if the developer of the app has set a certain boolean. An example of this is Example 2 in Section 3.1.1.

In Example 2, we would start tracking `imei` because the result of `getDeviceId()` is stored in it. Since `imei` is after that passed to `leak()`, a leak is reported. It is however clear to see that depending on the value of `someBooleanValue`, the developer might not have had the intention to leak the IMEI, and leaking it might never actually happen.

Tracking conditional statuses
To see whether it is possible that the use of a conditional results in the leaking of data, we need to know whether the value upon which the conditional switches can be a certain value. Let us stick with `if` statements for now. In the previous example, we want to know whether it is not the case that `someBooleanValue` is always set to `false`. If we find a place where `someBooleanValue` is set to `true`, or if we cannot determine what the variable is set to, we will assume it is set to `true`.

Example 7: Leaking based on a class field

```
1  function1:
2      this.someBooleanValue = true;
3  function2:
4      this.someBooleanValue = false;
5  function3:
6      if(this.someBooleanValue)
7          leak(privateInformation);
```

To figure out what values a variable can have, we need to track back from the place we are using the variable, in this case the if statement. While tracking back we look for instructions that assign values to the variable we are tracking. When an assignment to our variable is found, multiple things could happen: true is assigned, in which case we assume the variable is equal to true; false is assigned, in which case we assume the variable is equal to false, or something else is assigned. When something other than true or false is assigned to our variable, we either track whatever is assigned, or when we cannot track it we assume the variable is equal to true. In the case that different values could have been assigned to the variable (see Example 7) we choose true. The reason we choose true in Example 7 is because it is very hard to determine what the value of `this.someBooleanValue` is at the moment the leak occurs.

The problem with this approach is that tracking potential values of one variable can quickly branch into the tracking of multiple variables, which might not be booleans. Consider the following example:

Example 8: Leaking based on a comparison

```
1  int value1 = value2 + value3;
2
3  if(value1 < value4)
4      leak(privateInformation);
```

To determine, in the example, whether the leak can occur, we need to determine whether `value1` is smaller than `value4`. To determine this we already need to track two variables. When we start tracking these two variables we will eventually see the `value1 = value2 + value3` expression. To be able to figure out whether our condition can be true we now also have to track `value2` and `value3`. Because the amount of variables to track can be easily expanded, tracking the values of variables can potentially have quite an impact on performance. Another problem that occurs in this example is that we are no longer tracking booleans but integers or floats. Because we are now tracking integers and floats, we have to detect what possible values all the variables we track could have. Let us assume that `value2` could be either eleven or three, and that `value3` could be either one or seven, in this case `value1` could be twelve, four, eighteen or ten. If more branching occurs, the amount of values `value1` could have, can increase exponentially, which will also have a negative impact on performance. In the end it might be a better solution to assume that complex conditions, such as in the example, can be true. With this solution, we as-

sume that if code was meant to be possibly dead, a boolean would have been used to indicate this (such as `this.someBooleanValue` in Example 7).

5.3 Usage of source to sink paths

Once we have found paths that lead from source to sink, we need to ask ourselves when we will actually be leaking. Even though we have found a path from source to sink, we are not sure yet whether the code in this path is actually reachable. An app could, for example, have a function which acquires the phone's IMEI, creates a socket, and passes the IMEI to the socket. However, if the mentioned function is never called, in which case it is "dead code", the IMEI is never leaked.

To be able to detect whether the leak actually occurs, we will need to split the code in this path in several parts and see if every part in this path is executed. To see how to choose these parts let us look at a possible data path:

Example 9: Data path

```
1 function1:
2     this.imei = manager.getDeviceId();
3 function2:
4     String localImei = this.imei;
5     return localImei;
6 function3:
7     String localImei = function2();
8     leak(localImei);
```

In this example, we will start tracking in `function1` since that is where `getDeviceId()` is called. Since the IMEI is stored in `this.imei`, we will have to look for usages of `this.imei`. `this.imei` is used in `function2` where it is returned, causing us to search for usages of `function2`. `function2` is used in `function3` where the result of the function call is leaked. We cannot just simply look whether `function1` is called, because just a call to `function1` only implies that the IMEI is stored, not that it is leaked. We could check whether every function in the path is called. This approach would be sufficient to show that the path we have found can be executed. The problem with the last approach is that it is a little too complex, we do not need to check if all the functions are called, since calling `function3` implies calling `function2`.

To reduce this problem and understand it, we need to look at the control flow in this example. The problem with tracking data through an app is that the path data takes is not necessarily the same as the execution path of the app. In the given example, we can distinguish two different control flows. The first control flow is done in `function1`, because `function1` is not called by `function2` or `function3` and it does not call those functions either. Because `function1` encompasses a control flow, we need to check if `function1` is actually called. The second control flow starts with `function3`. After the control flow is started, it continues in `function2` because `function3` calls `function2`. Because the second control flow starts in `function3`, we also need to check whether `function3` is called. To generalise the last control flow, we could say that the function that starts the control flow is the function that does not return the data we are tracking, and the function that is not called by any of the other functions in the path.

Now that we know which functions need to be called for a leak to happen, we can find out if they are called. To find out if a function is called, we can make a list of all the functions that call this function. If the first function started in the app is in this list, we know our function is called. In Android the first function to be called is generally the `onCreate` of an activity, although more functions exists that could start an Android app. If the first function is not in the list, we repeat the process for the functions in the generated list. This is an example in pseudo-code of how this algorithm would work:

Example 10: `isFunctionCalled` algorithm

```
1 bool isFunctionCalled(function):
2     var functionList = function.isCalledBy()
3     if onCreate in functionList:
4         return true
5
6     for function in functionList:
7         return isFunctionCalled(function)
8
9     return false
```

6 Evaluation and results

We have tested Anaconda on a number of popular apps on the Android Market. Our selection of apps is comprised of popular apps that a large number of Android users will have installed, and apps for Dutch services that are also popular. We then analysed the results in further detail.

The different types of leaks are explained in greater detail in Appendix A. Throughout this section we use the simplified names for the leaks instead of the names of the methods we actually

track.

For some apps we made a distinction between *intended usage* and *unexpected leaks*. Some apps send out private information, but this is what the app is supposed to do. That would be intended usage. However, if an app sends out private information that it logically should not need in order to function, that would be an unexpected leak. We cannot detect if a leak is intended or unexpected; both are marked as leaks by Anaconda.

In the following sections we discuss some of the interesting leaks, behaviour of the app and false positives by Anaconda.

6.1 Analysed apps

6.1.1 Dropbox

Leaks:

Leak	Amount
Account user data	1
Query	5
Wifi MAC address	1
Network operator	1

We analysed the Dropbox app because it is a popular app to synchronise files between your smartphone and other devices. To use the app you need to log in to your Dropbox account, so it is expected to use account data like username and password.

The user's network operator is leaked via an HTTP GET to a URL on Dropbox's domain, `https://api.dropbox.com/1/report_host_info?`. The network operator name is sent to this address, along with other data such as the app version, the user's locale, the phone's model name and Android version. This is very clearly a leak of the user's personal information. It seems that this code is used to gather statistics on the app's users. This is an unexpected leak; gathering this data is not essential for the app to function. Because you need to log in to your Dropbox account to use the app, it might be possible that this information can be traced back to the specific user of the app rather than being anonymous.

The amount of obfuscation in the APK is quite high, which makes manual verification of the results challenging. We can say for certain that the app only uses the leaked data to create text strings using a `StringBuilder`, and only sends data to an address on their own domain, not to an external ad server.

6.1.2 Barcode Scanner (zxing)

Leaks: nothing.

Barcode Scanner is an app to scan barcodes and QR codes with the smartphone's camera and is an implementation of the ZXing library. Other apps can also use this app to scan barcodes, instead of having to write their own implementation.

We did not detect any leaks in this app. There are only a few sinks and sources found, and there is not a path between any of them.

The app also uses the Wifi SSID and WEPKeys. One of the features of the app is to read Wifi settings from a QR code, so that you can automatically connect with a network without having to type the password. We did not detect that this data is being leaked.

6.1.3 Humble Bundle Downloader

Leaks: nothing.

The Humble Bundle app allows you to easily download the games you have purchased through the Humble Bundle service. We did not detect any leaks in the Humble Bundle Downloader app. Again, only a few of the sources and sinks that we track were found, and none of the sources are leaked.

6.1.4 Tweakers

Leaks:

Leak	Amount
Network operator	1

Tweakers.net is a Dutch technology-related website. The Tweakers app allows you to view the site's contents with an Android-native interface.

The Tweakers app is mostly clean; few of the sources and sinks that we track have been detected, in other words the app does not even *use* most of your private information, let alone leak it. However, the app does leak your network operator. This happens in a method that appears to fill a hash map with various types of information such as the size of your screen, the Android version and language and a number, and other ad-related information retrieved from the `com.google.ads` (AdMob) package such as the session id and the position an ad should appear.

In the end, a `StringBuilder` takes all the information from this hashmap and concatenates it into a single string. What is done with this resulting string is unclear, but it will probably be used

to retrieve ads targeted to the specific user's device and preferences.

6.1.5 Sudoku Free

Leaks:

Leak	Amount
Account name	1
Query	1
Last known location	6
Wifi MAC address	2
OS serial	1
IMEI	5
Phone number	1
Network country iso	1
Network operator	5
Sim country iso	2

The Sudoku Free app has a staggering list of detected leaks. Closer inspection reveals that many of these leaks come from the fact that Sudoku Free uses multiple ad APIs at the same time. Among the list of sources we track, the APIs that leak these sources were:

- Millennial Media
- AppLovin
- Mobclix
- MoPub
- Tapjoy
- Admob

Each of these APIs leaks some piece of private information, with some overlap in what sources they leak. Only one leak is caused by code from the app's developers themselves. This was the IMEI. The developer seems to have his own package to manage its ads and user information. This package has a custom method `com.genina.util.version.VersionSpecificFunctionFacade.getDeviceId()` for getting the IMEI, and also stores the IMEI in a field.

This method is eventually used, through another method `getAndroidID()` that returns the IMEI from this field, to create a string `PHONE_MODEL` that contains the Android model and IMEI. This ID is finally leaked in a method `submitStackTraces()`. Apart from this, the developers do not leak any data themselves. It should also be noted that the developers have a webpage on their policy for protecting private information [15]; either they are unaware of the leaks or are breaching their own privacy guidelines.

Also interesting is to see which ad APIs acquired tracked sources, but were not detected as leaking. These were:

- Greystripe
- Pontiflex
- Inneractive
- Kiip

Aside from these, it is possible the app might use more ad APIs that did not even use the sources that we track.

6.1.6 Reddit Sync

Leaks:

Leak	Amount
Query inputstream	1
Query	2
Network operator	1

Reddit is a social news site. The Reddit Sync app allows you to browse the site with a more mobile-friendly interface.

The network operator is stored in a hash map and then passed to the Admob API. A query inputstream is used in decoding bitmap images, and because this was marked as a sink somewhere else in the app, it is seen as leaking. Overall there is not much interesting happening in the app.

6.1.7 Word Search

Leaks:

Leak	Amount
On location changed	1
Last known location	7
IMEI	3
Network operator	2

Word Search is an app to play, as the name suggests, a word search game. As with Sudoku Free, this app uses six different ad APIs. The cause of the leaks originates from these APIs. Three of them leak the user's last known location, each in their own way.

6.1.8 gReader

Leaks:

Leak	Amount
Account name	2
Query	28
Network Operator	1

The network operator is leaked via Google Admob. Query is leaked many times, and called without leaking even more often. Interestingly, AndroidQuery, a library for manipulating UI elements, leaks Account Name. There does not seem to be a reason for a UI library to request an account name from the device, so this is very odd.

6.1.9 Shazam

Leaks:

Leak	Amount
Query	19
On location changed	1
Last known location	3
OS fingerprint	2
IMEI	5
Network country iso	1
Network operator	4
Subscriber id	1

Shazam is an app that allows the user to identify music by using the microphone. For this app, it makes sense that it connects to a remote server to do the identification process, but for this it should not require much private information. The app does use the last known location in order to share where you have heard a song with your friends, so that is intended usage. As before, Shazam uses a number of ad APIs that leak the usual bits of private information such as the user's location and the IMEI.

Of interest is the fact that Shazam leaks the IMEI and IMSI (device id and subscriber id), along with some other irrelevant information, into a custom configuration class that had been marked as a sink. The app also uses two libraries for crash reporting, one of which, Crittercism, also leaks the IMEI.

Some library, which is unidentifiable due to obfuscation, leaks the network operator. It is not likely to be code written by the app's developers, because the rest of the app's code is not obfuscated.

6.1.10 Buienradar

Leaks:

Leak	Amount
Query	1
On location changed	1
Last known location	2
OS serial	1

Buienradar is a Dutch app that displays the weather. The app can also show the weather in your current area if you enable it. As such, it is logical that the app can "leak" your current location, because it uses your location to check the weather in the area. This is intended usage. However, the source "on location changed" is leaked by Adgoji, an ad API, which has methods like `setLongitude` and `setLatitude`; it is obvious that Adgoji is also requesting and leaking the user's location aside from Buienradar itself.

An analytics API, comScore, is used in a very long track that leaks the last known location, so it is possible that it is being leaked by that API too.

Also interesting is that the GSM CID and GSM LAC are also being used (though not leaked). These methods return the cell id, a unique number for the mobile network cell the phone is located in (in other words which network tower the phone is connected to), and the location area code. These numbers are specific to the user's network operator, and can be used to very roughly determine where the user is located. The cell radius for a network can range from hundreds of meters to multiple kilometres, so it is not going to be very precise. Another problem with this method is that it requires a database of cell ids and area codes, which the network operators do not officially supply. There are unofficial databases [16], but those will not have perfect accuracy. This is the only app in our test set that uses these methods.

6.1.11 Twitter

Leaks:

Leak	Amount
Account name	1
Account password	1
Account user data	7
Bluetooth address	1
Query inputstream	3
Query	4
On location changed	1
Wifi MAC address	1
IMEI	1

The leaking of password and account name is probably intended usage, because the app needs to send your username and password to its authentication servers. From the results it seems like the account name and password are indeed being used in this way.

The app uses a crash reporter library called Crashlytics, which leaks the Bluetooth address. However, to call this method the app requires the `BLUETOOTH` permission, which the app does not request. The app cannot actually leak the Bluetooth address in this way because of this.

Twitter also has a feature "Tweet Location", which adds geolocation data to your tweets. This setting is off by default, but Anaconda tracks it anyway regardless of this setting. The app leaks "on location changed", and uses but does not leak the last known location. It is unclear due to obfuscation which of these two methods the app uses for geolocation when you send a tweet. It seems more likely to request the last known location upon sending a tweet, so it is more likely that that method is used. That raises the question what "on location changed" is being used for.

6.1.12 WhatsApp

Leaks:

Leak	Amount
Account name	5
Account type	6
Query inputstream	7
Query	38
On location changed	2
Wifi MAC address	1
IMEI	2
Network operator	3
Sim operator	1

WhatsApp is one of the larger results, with 65 instances of leaks, 38 of which come from leaking query. This also resulted in the largest HTML page being generated by Anaconda, at 421 MB in size.

The Wifi MAC address is leaked by the PayPal API, an online payment system, by appending it into a string. PayPal also leaks the IMEI into a string.

6.1.13 iGroningen

Leaks:

Leak	Amount
Account name	1
Account password	1
Account user data	1
Bluetooth address	1
Query inputstream	2
Query	9
Last known location	2
Wifi MAC address	1
IMEI	3
Phone number	1
Network country iso	1
Network operator	4
Sim country iso	1
Sim operator	2

iGroningen, an app which gives University of Groningen related information to students, leaks even more than Sudoku Free; 30 leaks in total. It is quite difficult to analyse the results manually due to the heavy obfuscation and the large size of the app.

Particularly strange is the fact that the app uses and leaks the phone number. What this method returns is very device-dependant, but typically it returns the main phone number (phone number for line 1) on the device. This depends on if the phone number is stored on the SIM card, and might fail depending on device and provider. Nevertheless, the app seems to use the phone number, along with the device name, carrier, OS version and other information in some login-related code.

The leak results for the account user data is a good example of a false positive result given by Anaconda. The leak in this case is found in a method that makes the following call:

```
1    p7.equals(this.b.getUserData(v0, "school_id"))
         == 0
```

Here, `p7` is a string passed to this method and `v0` is the data being leaked. In this case, the user data `school_id` is retrieved and compared with the given string. String comparison obviously does not leak data out of the app. The reason this method leaks is because `p7` has been marked as a sink somewhere in the app. It is difficult to find where and why exactly this was, but in the end it is clear that the method does not actually leak.

6.1.14 NS Reisplanner Xtra

Leaks:

Leak	Amount
Account name	1
On location changed	2
Last known location	1

The NS is the "Nederlandse Spoorwegen", the main railway operator in the Netherlands. Their Android app allows the user to plan their journey and check the departure times of trains per station.

The NS app also uses the AndroidQuery library that gReader uses. In this app, it leaks the account name again, just like before.

This app requests the `ACCESS_FINE_LOCATION` permission. For a travel planning app this makes sense: for example, a user might want to find the earliest leaving train at the train station closest to his current location. For the most part it seems like the app is keeping track of the user's location in order to use it elsewhere in the app. However, "on location changed" is overridden by AndroidQuery,

which uses it to print the user's location to a log every time the location changes. This is not as bad as sending the location to a server over the internet, but once the location is in the log anything could happen with it, for example, writing it to a file or sending it over the internet later once the app closes.

6.2 Resulting HTML

Anaconda outputs its results to an organised HTML file. Figure 5 shows what this webpage looks like.

On the left side of the page is a bar with the list of sinks and the list of sources. In this bar, it can be seen that the network operator, query inputstream and query are being leaked by the app. Currently the tree track for `Landroid/content/ContentResolver;->openInputStream` is selected.

At the top right part of the screen the tree for this track is displayed. Here there are four nodes in this track tree. Currently the code for the node `b()Vstart-register:v1` is selected. This can be seen from the method name above the Java source code. The comments for the top level node `a(Landroid/net/Uri;)Vstart-register:v?` indicates that query inputstream is being leaked into the field `M` of the class `Lp/ag;`. The three nodes below the top level node in the tree use this field.

The currently selected node can leak the data from the field `M`, which is indicated by the text "Leaks:" appearing before the method name in the tree. The calls that use the field `M` can be seen in the Java source code for this method and in the annotated bytecode instructions. Here, the instruction for `this.V.setImageBitmap(this.M);` can leak query inputstream through the field `M`. This is indicated by the comment "Data is put in sink!" in the bytecode instructions.

6.3 Discussion of results

Figure 6 shows the amount of times each type of leak was detected in the apps that we analysed. In general, many apps use and leak query. This method can be used to retrieve data such as your contacts and call log, but it could be used in a harmless way as well. We cannot track exactly what this method is being used for in an app; we discuss this further in Section 7.5.

The table in Appendix D shows the number of times a source was leaked and the number of times a source was requested for each of the apps. The total number of leaks over the total number of access can be seen as an indicator of how many times the apps in general leak private information. Any time a source is requested, it could potentially be leaked too.

Other popular leaks are user's account names. This can be explained by the fact that many of these apps allow the user to log in to the service; it makes sense that an account name stored on the device is being sent to, for example, an authentication server, which is intended usage.

We also analysed what kind of calls are made by the various ad APIs used by these apps, and how many times these calls result in leaks. The results can be found in Table 2. The most common leaks by these APIs are Last Known Location, Query, Network Operator and IMEI. Developers often use the IMEI as a unique identifier for the user. For example, some apps add it to the list of parameters when making a request to a URL.

Anaconda does not track if the methods it detects as leaking are actually called by the app. An app could have defined a method that requests and leaks the IMEI, but if the app never calls that method your data is not being leaked, even though Anaconda would detect that the app leaks.

6.4 Analysis time

In this subsection we will discuss the time efficiency of Anaconda. The measured time consists of only the time to analyse, not the time to decompile the app and the time to generate the HTML file.

Figure 7 plots the time to analyse the app vs. the size of the app (the size of the `classes.dex` file in the APK). The y-axis of this plot is in a logarithmic scale. It looks like the analysis time increases linearly with the app size, but there is too much variance in the data to get a proper conclusion from this data.

Figure 8 is a similar graph that plots the analysis time vs. the number of sources and sinks used by the app. Here it can be clearly seen that analysis time increases linearly with the number of sources and sinks, as is to be expected. There is an interesting difference between gReader and WhatsApp: gReader has many more sinks and sources, but a shorter analysis time. An explanation for this could be that gReader uses sources and sinks many times, but the actual leak tracks are short resulting in a short analysis time.

Figure 5: Full-sized screenshot of the HTML report

Anaconda Report

SINKS

Ljava/net/URL; created
Ljava/net/URL; created
Ljava/net/URL; created
Ljava/io/FileOutputStream; created
Ljava/io/FileOutputStream; created
Ljava/io/FileOutputStream; created
Lorg/apache/http/impl/DefaultHttpClientConnection; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created
Lorg/apache/http/impl/client/DefaultHttpClient; created

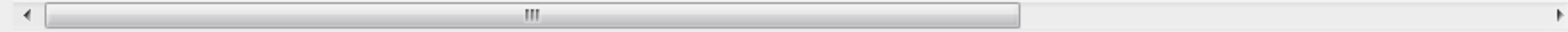
SOURCES

Landroid/telephony/TelephonyManager;->getDeviceId is called
Leaks: Landroid/telephony/TelephonyManager;->getNetworkOperator is called
Leaks: Landroid/content/ContentResolver;->openInputStream is called
Landroid/content/ContentResolver;->openInputStream is called
Leaks: Landroid/content/ContentResolver;->query is called
Landroid/content/ContentResolver;->query is called
Leaks: Landroid/content/ContentResolver;->query is called

a(Landroid/net/Uri;)V

Tree:

- Branch leaks: a(Landroid/net/Uri;)V start-register: v?
  - Leaks: b()V start-register: v1
  - Leaks: run()V start-register: v1
  - Leaks: a(Ljava/lang/String;)V start-register: v1



Comments:

Lp/ag;->a(Landroid/net/Uri;)V start-register: v?
Lp/ag;->b()V start-register: v1

```
public final void b()
{
    v0 = new java.io.File(this.L.getPath());
    this.M = android.graphics.BitmapFactory.decodeFile(v0.getPath(), k.f.a(v0));
    this.V.setVisibility(0);
    this.V.setImageBitmap(this.M);
    this.X = 3;
    return;
}
```

```
0    new-instance['v0', 'Ljava/io/File;']
     iget-object['v1', 'v2', 'Lp/ag;', 'L', 'Landroid/net/Uri;']
     invoke-virtual['v1', 'Landroid/net/Uri;', 'getPath()Ljava/lang/String;'] (marked as sink)
     move-result-object['v1']
     invoke-direct['v0', 'v1', 'Ljava/io/File;', '<init>(Ljava/lang/String;)V'] (marked as sink)
     invoke-virtual['v0', 'Ljava/io/File;', 'getPath()Ljava/lang/String;'] (marked as sink)
     move-result-object['v1']
     invoke-static['v0', 'Lk/f;', 'a(Ljava/io/File;)Landroid/graphics/BitmapFactory$Options;']
     move-result-object['v0']
     invoke-static['v1', 'v0', 'Landroid/graphics/BitmapFactory;', 'decodeFile(Ljava/lang/String; Landroid/graphics/BitmapFac
tory$Options;)Landroid/graphics/Bitmap;']
     move-result-object['v0']
     iput-object['v0', 'v2', 'Lp/ag;', 'M', 'Landroid/graphics/Bitmap;']
     iget-object['v0', 'v2', 'Lp/ag;', 'V', 'Landroid/widget/ImageView;']
     const/4['v1', '0']
     invoke-virtual['v0', 'v1', 'Landroid/widget/ImageView;', 'setVisibility(I)V']
     iget-object['v0', 'v2', 'Lp/ag;', 'V', 'Landroid/widget/ImageView;']
     iget-object['v1', 'v2', 'Lp/ag;', 'M', 'Landroid/graphics/Bitmap;']
     invoke-virtual['v0', 'v1', 'Landroid/widget/ImageView;', 'setImageBitmap(Landroid/graphics/Bitmap;)V'] (marked as sink)
         |
         |------> ++v1: Tracking register (first register tracked in this method)
         |------> v1: Data is put in sink!

     const/4['v0', '3']
     iput['v0', 'v2', 'Lp/ag;', 'X', 'I']
     ->
1        return-void['']
         ->
2            move-exception['v0']
             invoke-virtual['v0', 'Ljava/lang/Exception;', 'printStackTrace()V']
             goto['-5']
             ->
                 Go to: 1
```

Lp/an;->run()V start-register: v1
Lp/ag;->a(Ljava/lang/String;)V start-register: v1

© Anaconda

Table 2: Amount of times ad-APIs acquire and leak private information per app. a/b means: a out of b acquired pieces of information were leaked.

	query	**network operator**	**on location changed**	**location**	**Wifi MAC address**	**IMEI**	**phone number**	**network country ISO**	**SIM country ISO**	**account name**	**OS fingerprint**	**OS serial number**	**SIM operator**	***total***
tweakers														
admob	0/2	1/1												1/3
sudoku free														
admob	0/2	1/1												1/3
mobclix	0/1	0/1	0/3	2/4		1/1					0/2			3/12
inneractive			0/1	0/1		0/1								0/3
millenialmedia	1/1			1/1		1/1								3/3
mopud	0/1			2/2										2/3
applovin		1/1			1/1	1/1	1/1		2/2	1/2		1/1		8/9
tapjoy		3/6			1/1	1/1		1/1						6/9
greystripe	0/1					0/1	0/1							0/3
portiflex							0/1							0/1
kiip		0/1		0/1										0/2
word search														
jumptap		1/1	1/1	1/1										3/3
adfonic		0/2		0/2									0/1	0/5
inmobi	0/2			2/2										2/4
mobfox				4/4		2/2								6/6
millenialmedia						1/1								1/1
admob	0/2	1/1												1/3
greader														
admob	0/2	1/1												1/3
shazam														
flurry			1/1	1/1		1/1								3/3
admarvel	1/5		0/1	0/1										1/7
sessionm		2/4		2/2		2/2		1/1						7/9
admob	0/2	1/1												1/3
buienradar														
adgoji		0/1	1/1	0/1		0/1						1/1		2/5
reddit sync														
admob	1/2													1/2
total														
*	3/23	12/22	3/8	16/24	2/2	10/13	1/3	2/2	2/2	1/2	0/2	2/2	0/1	53/105

Figure 6: Histogram of the number of leaks in our test set

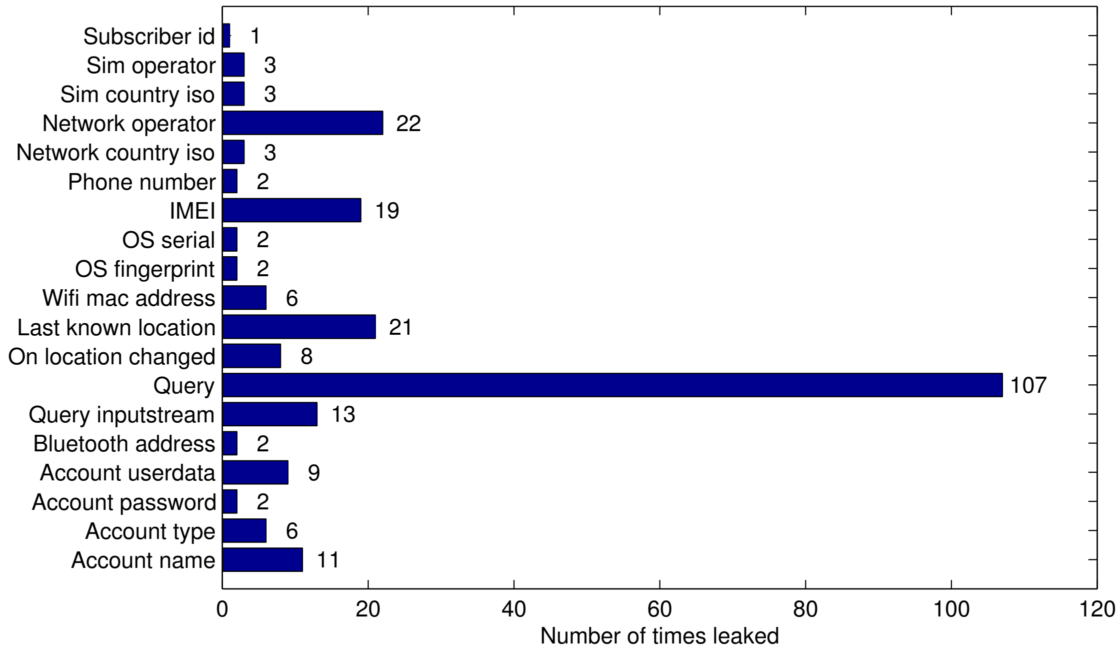
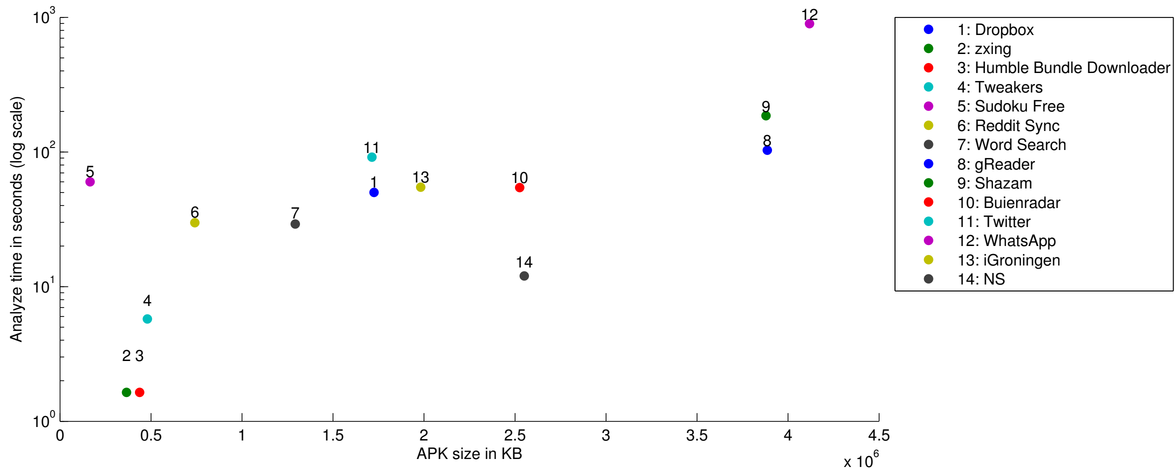


Figure 7: Analysis time vs. app size



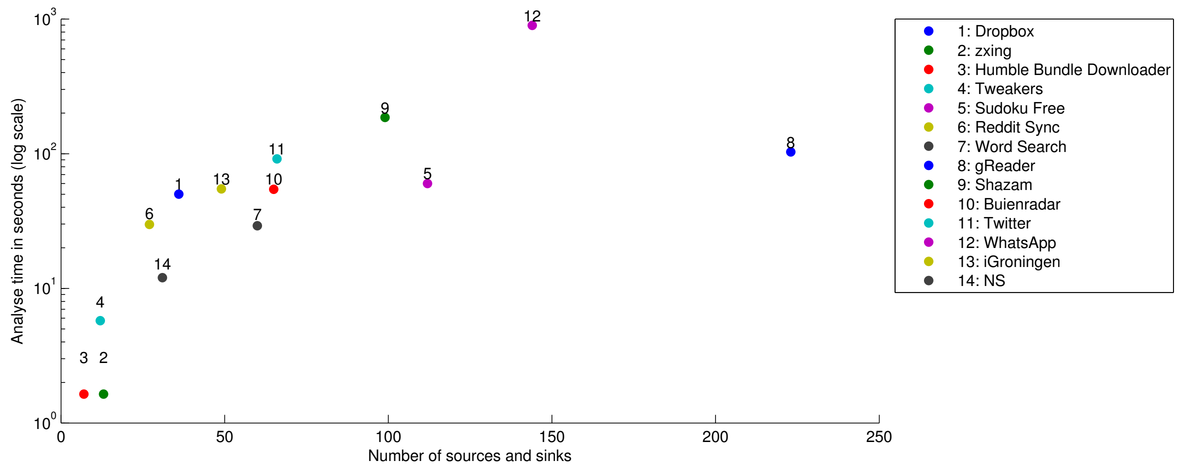
7 Future work

There are still many possibilities to improve our analyser, some of which have already been described in Section 5. Static code analysis is a large field, with a great number of possibilities for enhancements. As a result, more improvements can always be made to find more leaks, reduce the number of false positives or optimise the analysis to make it faster. Some ideas we came up with are discussed here and could be investigated further in the future.

7.1 Inheritance

While we do provide support for inheritance in the case of function calls while tracking, we are currently not looking for any subclasses of the sources,

Figure 8: Analysis time vs. number of sources and sinks



sinks, fields, or listeners we search for. Androguard unfortunately does not provide functionality to do this. It should, however, be relatively simple to extend our current implementation which finds subclasses to make it usable for these other cases. It would allow our system to detect cases where, for example, a Socket is subclassed, after which information is send by calling that class' send methods. These methods then call the Socket class' send methods, leaking the information.

7.2 Permissions

In the results, we have seen many apps using ads. As we have shown, many of the ad APIs provide functions which need to be called before data is leaked, which set a boolean for example. As we are currently unable to detect whether or not these functions are called, and thus assume they are, false positives are introduced. A way to reduce the number of false positives is to check whether or not the required permission is requested by the app. If, for example, the ad API is able to leak your location, but only does so when the developer calls a function to set a boolean to true, a permissions is required to request this information. If the permission is missing, it is impossible for the app to be leaking it. This could be used to filter the leaks depending on whether the required permission is requested or not.

7.3 Reflection

Anaconda has no support for reflection. The simplest solution would be to assume all functions called through reflection which take private information as a parameter will leak it or return it if they do not. There are however many legitimate usages for reflection in Android, the most important one being backward compatibility. A better solution might be to try and resolve the reflection. While this is a very difficult problem in static analysis, it is theoretically possible to solve when the strings used are available in the code. These strings could be used to look up the real function that is called. The strings could, however, be obfuscated, as shown in Example 1, or even send to the device over the internet. In these cases it would be impossible to determine the function being called, but it also almost guarantees something malicious is being done with the information.

7.4 Intents

As explained in Section 2, intents are messages used to communicate between processes, either within the same app or with another app installed on the phone. Information send with intents is hard to keep track of. At compile time the receiver may not even be known, for example, in the case of a "ACTION_SEND" intent a receiver is selected by the user. While it is very uncommon to send private information this way, it could be a good solution to leak private information for malicious app developers trying to avoid detection. When the receiver is explicitly specified and is part of the app being analysed, this could be tracked in theory. The receiving activity or service can be determined, leading to the function called when the intent is received. When the function is known, tracking could continue there. This would lead to less false positives than marking any intent containing private

information as leaking.

7.5 Content providers

Another feature of Android that we could analyse further to reduce the amount of false positives is the content provider. A content provider is a interface for data, where the data can be in a number of forms like files or a database. It is often used to share data among apps. Information can be retrieved by performing a query on the content provider using a content resolver.

Android provides numerous content providers, among which are content providers containing your contacts, your call history and your calendar. Clearly they hold private information. To recognise if this information is accessed, we look for queries on content providers. We are, however, unable to detect which content provider is referenced. This leads to false positives. To reduce the number of false positives, we could attempt to detect which content provider is accessed. As they are references by a constant value, this should in most cases be possible to detect. It does however require backtracking to find out which content provider is accessed. We could then only track the queries on the content providers containing private information.

8 Related work

8.1 Androguard

Androguard [6] is a combination of a set of tools for analysing Android apps, written in Python. One of the things among what Androguard does is decompiling Android package files (APKs). It can disassemble dex files with a number of external tools, or use its own decompiler called DAD (DAD is a decompiler), which is included in Androguard and is also written in Python. Androguard parses these dex files and forms readable instructions. These are put in a structure of classes, methods and blocks. It then provides an API, which can then be used to build your own analyser.

Androguard does not do much analysis by itself. It does however provide you with functionality to directly search the APK. This allows you for example to look up usages of methods or fields. Other tools like Androwarn [9] and Andrubis [10] use this to detect usages of known sources of private information, like telephone identifiers or the location of your phone. They then assume the information is used in a malicious way. This is because Androguard does not track information through the app. While using private information does suggest that the app could leak the information, it does not guarantee this information will actually leave the phone, or the code in which it happens is even called. Especially in ad-APIs it is not uncommon to have functions that could leak private information, but only do so when called in the code of the developer.

Anaconda goes a step further. Besides just checking if and where a certain method is called, we then look for a path from this source of information to a location where it leaves the phone, like a socket. When such a path exists, we know this information can be leaked. This process is explained in more detail in Realisation (Section 4).

8.2 TaintDroid

While Anaconda is a static analysis tool, it is also possible to analyse an Android app dynamically. This dynamic analysis has been done with a tool called TaintDroid [7]. TaintDroid is an extension to Android which modifies Android's virtual machine so it can track tainted data as it flows through the app. The tracking of data is accomplished by giving every piece of data a taint-tag, stating whether or not the data is tainted with private information. The nice thing about this dynamic approach is that it is very precise and it can find leakages that are otherwise hard to detect, for example, when a leaking method is called through reflection or when tainted data is passed from one app to another app. The disadvantage of such a system is that it takes a toll on CPU and RAM usage at app runtime, in the case of TaintDroid: about 14% CPU overhead and about 4.4% RAM overhead. Another disadvantage is that such a system can only track the data-flow of an app and not the control-flow, whereas static analysis is capable of also tracking the control-flow.

8.3 SAAF

SAAF [11] (Static Android Analysis Framework) is a static analysis tool that uses program slicing to cut away irrelevant parts of the APK, and then analyses the remaining code for patterns, to detect if an app is leaking private information. Like Anaconda, SAAF also analyses smali code. SAAF looks for certain method usages and uses data flow analysis on the parameters of the methods. By tracking parameters back SAAF can determine whether these are constant values, which is useful to determine risk factors of certain function calls. For example, a function could be executing a harmless call to “ls”, or it might be trying to gain superuser rights with a call to “sudo”. SAAF is thanks to this also capable of detecting if, for example, an app is

trying to send text messages to a hard-coded SMS number, and SAAF can also give you this number.

Although SAAF does some data-flow analysis, it does not try to determine whether actual leaking of private information occurs. Because of this the actual results produced by SAAF are more comparable to the results of tools like Androwarn [9], than to the results of Anaconda.

8.4 Stowaway

Stowaway [17] analyses the permissions an app uses and checks whether or not the app is overprivileged. Requesting more permissions than necessary may increase the impact of a bug or a vulnerability in the app. To find out which permissions are really needed it statically analyses the app and searches for API calls. It then compares these API calls with the required permissions. This however is not as trivial as it sounds. Android documentation regarding permissions is limited. Only 78 methods for Android 2.2 are documented to need permissions, while Felt et al. found 1259 API calls to generate a permission check. They constructed a map from this data resulting in a more complete list of required permissions.

9 Conclusion

Many apps have access to private information on your smartphone, but it is often unclear what exactly is done with it. Even when there is a clear legitimate use for it, there is no guarantee it is not leaked to a third party. With Anaconda we have shown that many apps leak private information to remote servers. While in some cases this is to be expected due to the functionality of the app requiring the data, in many cases this is clearly not the case. We found that 7 of the 14 apps we tested to contain ads, of which each and every single one was requesting and leaking private information. For 243 of the 572 requests of private information found by Anaconda, a path was found from the source to a sink, indicating the information leaves the system.

With the results given by Anaconda, we have shown that static code analysis is a viable way to detect private-information leaks in apps. Anaconda already detects numerous potential leaks, and suggested future improvements will most probably only increase the amount of detected leaks. All in all, Anaconda has proven itself to be a useful tool for analysing Android app behaviour and for detecting leaks in apps.

References

[1] Gartner. Gartner Says Asia/Pacific Led Worldwide Mobile Phone Sales to Growth in First Quarter of 2013. `http://www.gartner.com/newsroom/id/2482816`.

[2] WhatsApp is broken, really broken. `http://fileperms.org/whatsapp-is-broken-really-broken/`, September 2012.

[3] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *TRUST'11 Proceedings of the 4th international conference on Trust and trustworthy computing*, pages 93–107, 2011.

[4] Frank Maker and Yu-Hsuan Chan. A Survey on Android vs. Linux. Technical report, University of California, 2009.

[5] Gabor Paller. A list of Dalvik opcodes. `http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`.

[6] Anthony Desnos. Reverse engineering, Malware and goodware analysis of Android applications. `https://code.google.com/p/androguard/`.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, Anmol N. Sheth, and Leslie Lamport. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[8] CERT. Static source code analysis tools. `https://www.cert.org/secure-coding/tools.html`.

[9] Thomas Debize. Androwarn - Yet another static code analyzer for malicious Android applications. `https://github.com/maaaaz/androwarn`.

[10] Andrubis: A Tool for Analyzing Unknown Android Applications. `http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/`.

[11] Johannes Hoffmann, Martin Ussath, Michael Spreitzenbarth, and Thorsten Holz. Slicing Droids: Program Slicing for Smali Code. In *SAC'13*, March 2013.

[12] Kostyantyn Vorobyov and Padmanabhan Krishnan. Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches. In *SSV '10*, 2010.

[13] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *23rd Annual Computer Security Applications Conference*, 2007.

[14] J. Veldthuis, S. Groenewold, and K. L. Winter. Anaconda. `https://github.com/KPWhiver/Anaconda`, 2013.

[15] Genina. Privacy Policy for genina.com. `http://www.genina.com/pages/privacy/privacy.jsf`.

[16] OpenCellId. `http://opencellid.org/`.

[17] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *CCS '11 Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

Appendix

A Leak type glossary

This is a list of terms used to describe the different kinds of data that was acquired by apps. Please note that this list only includes data that was actually found by Anaconda as being acquired, other possible kinds of data exists, but these were never acquired.

Query: A way for an app to get information from a local databases stored on the device. These databases contain, amongst other things: stored SMS and MMS messages, contacts, bookmarks, and the call log.

Query inputstream: Allows an app to get the same data as with a query, only through an `InputStream`.

Wifi MAC address: The MAC (media access control) address is a unique number associated with the Wifi adaptor of a phone.

Wifi SSID: The SSID (service set identifier) is the name of a Wifi network that the phone can connect to. This information can be accessed by requesting a list of configured Wifi networks, and looking for this value in that list.

Wifi WEP keys: WEP (wireless equivalent privacy) keys that are used to access configured Wifi networks on the phone. Just like the SSID, this information can be accessed from a list of configured networks.

Bluetooth address: The MAC address of the phone's Bluetooth adaptor.

OS serial: A hardware serial number.

OS fingerprint: A string that uniquely identifies this build of the OS.

Account user data: Data associated with a user account on the phone. Google has not clearly defined what this data should encompass. The user account of this request, and all other account related requests, could be an account made by this application, but it could also be, for example, a Facebook or a Google account.

Account password: The password associated with a user account on the phone.

Account type: The type associated with a user account on the phone.

Account name: The name associated with a user account on the phone.

On accounts updated: A listener that is passed accounts as soon as they are updated, e.g., when a new account is added.

On location changed: A listener that is passed the location of the phone whenever a new location is found, e.g., through GPS or GSM triangulating.

Last known location: The last known location of the phone.

IMEI: The IMEI (international mobile station equipment identity) is a unique number associated with the phone, as such it can be used to identify the phone.

Phone number: The phone number associated with the phone.

Network operator: The company that provides you access to the current mobile network, for example, T-Mobile and AT&T.

Network country ISO: The country of which the phone is currently using the network.

SIM country ISO: The country in which the SIM's (subscriber identity module) provider resides.

SIM operator: The company that provided the phone's SIM.

GSM CID: The GSM CID (cell ID) is a unique number referring to the base transceiver station, which facilitates communicating between the phone and a network.

GSM LAC: The GSM LAC (location area code) corresponds to a set of base transceiver stations. The base transceiver station identified by the GSM cid is part of this set.

On call state changed: A listener that is passed information when: the phone is being called, the phone is off the hook, or the phone is idle. If possible, an incoming phone number is also passed.

On cell location changed: A listener that is passed information about the new GSM cid and GSM lac whenever it changes.

Subscriber id: This is equivalent to the IMSI (international mobile subscriber identity). The IMSI is a number associated with the network the phone is allowed access to.

B List of recognised sinks

```
1  # These functions return sinks
2  java.net.DatagramSocketImplFactory -> createDatagramSocketImpl
3  java.net.http.AndroidHttpClient -> newInstance
4  android.net.http.HttpResponseCache -> install
5  android.net.http.HttpResponseCache -> getInstalled
6
7  javax.net.ssl.SSLSocket -> getOutputStream
8
9  java.net.URLStreamHandler -> openConnection
10 java.net.URL -> openConnection
11
12 java.nio.channels.ServerSocketChannel -> accept
13 java.net.ServerSocket -> accept
14 java.nio.channels.DatagramChannel -> socket
15 java.nio.channels.DatagramChannel -> connect
16 java.nio.channels.DatagramChannel -> disconnect
17 java.nio.channels.DatagramChannel -> open
18 java.nio.channels.SocketChannel -> open
19
20 java.sql.DriverManager -> getConnection
21
22 org.apache.http.conn.ClientConnectionRequest -> getConnection
23 org.apache.http.impl.conn.SingleClientConnManager -> getConnection
24 org.apache.http.impl.conn.AbstractClientConnAdapter -> getWrappedConnection
25
26 java.net.Socket -> getOutputStream
27 java.net.SocketImpl -> getOutputStream
28 android.bluetooth.BluetoothSocket -> getOutputStream
29
30 android.nfc.NfcAdapter -> getDefaultAdapter
```

```
31
32 # These classes are sinks
33 java.net.DatagramSocket
34 java.net.DatagramSocketImpl
35 java.net.MulticastSocket
36
37 java.io.FileOutputStream (indirect sink)
38 java.io.FileWriter (indirect sink)
39
40 java.net.HttpURLConnection
41 java.net.URLConnection
42
43 android.net.http.AndroidHttpClient
44
45 java.nio.channels.DatagramChannel
46 java.nio.channels.SocketChannel
47
48 javax.net.ssl.HttpsURLConnection
49
50 org.apache.http.impl.AbstractHttpClientConnection
51 org.apache.http.impl.AbstractHttpServerConnection
52 org.apache.http.impl.DefaultHttpClientConnection
53 org.apache.http.impl.DefaultHttpServerConnection
54 org.apache.http.impl.SocketHttpClientConnection
55 org.apache.http.impl.SocketHttpServerConnection
56
57 org.apache.http.impl.client.DefaultHttpClient
58 org.apache.http.impl.client.AbstractHttpClient
59 org.apache.http.impl.client.DefaultRequestDirector
60 org.apache.http.impl.client.BasicResponseHandler
61
62 org.apache.http.impl.conn.AbstractClientConnAdapter
63 org.apache.http.impl.conn.AbstractPooledConnAdapter
64 org.apache.http.impl.conn.tsccm.BasicPooledConnAdapter
65 org.apache.http.impl.conn.SingleClientConnManager.ConnAdapter
66 org.apache.http.impl.conn.DefaultClientConnection
67 org.apache.http.impl.conn.DefaultClientConnectionOperator
```

C List of recognised sources

```
1 # These methods are sources
2 android.accounts.AccountManager -> getPassword
3 android.accounts.AccountManager -> getUserData
4
5 android.accounts.AccountManagerService -> getPassword
6 android.accounts.AccountManagerService -> getUserData
7
8 android.accounts.IAccountManager$Stub$Proxy -> getPassword
9 android.accounts.IAccountManager$Stub$Proxy -> getUserData
10
11 android.accounts.Account -> toString
12
13 android.bluetooth.BluetoothAdapter -> getAddress
14 android.bluetooth.BluetoothAdapter -> getName
15 android.bluetooth.BluetoothDevice -> getName
16 android.bluetooth.BluetoothDevice -> getAddress
17 android.bluetooth.BluetoothDevice -> toString
18
```

```
19 android.bluetooth.IBluetooth$Stub$Proxy -> getAddress
20 android.bluetooth.IBluetooth$Stub$Proxy -> getName
21 android.bluetooth.IBluetooth$Stub$Proxy -> getRemoteName
22
23 android.server.BluetoothService -> getAddress
24 android.server.BluetoothService -> getAddressFromObjectPath
25 android.server.BluetoothService -> getName
26 android.server.BluetoothService -> getRemoteName
27
28 android.location.LocationManager -> getLastKnownLocation
29 android.location.ILocationManager$Stub$Proxy -> getLastKnownLocation
30
31 android.telephony.gsm.GsmCellLocation -> getCid
32 android.telephony.gsm.GsmCellLocation -> getLac
33 android.telephony.gsm.GsmCellLocation -> getPsc
34 android.telephony.gsm.GsmCellLocation -> toString
35
36 android.telephony.cdma.CdmaCellLocation -> getSystemId
37 android.telephony.cdma.CdmaCellLocation -> getNetworkId
38 android.telephony.cdma.CdmaCellLocation -> getBaseStationLongitude
39 android.telephony.cdma.CdmaCellLocation -> getBaseStationLatitude
40 android.telephony.cdma.CdmaCellLocation -> getBaseStationId
41 android.telephony.cdma.CdmaCellLocation -> toString
42
43 android.telephony.NeighboringCellInfo -> getCid
44 android.telephony.NeighboringCellInfo -> getLac
45 android.telephony.NeighboringCellInfo -> getPsc
46 android.telephony.NeighboringCellInfo -> getRssi
47 android.telephony.NeighboringCellInfo -> toString
48
49 com.android.internal.telephony.ITelephony$Stub$Proxy -> getCellLocation
50
51 android.net.wifi.WifiConfiguration -> toString
52
53 android.net.wifi.ScanResult -> toString
54
55 android.net.wifi.WifiInfo -> getBSSID
56 android.net.wifi.WifiInfo -> getIpAddress
57 android.net.wifi.WifiInfo -> getMacAddress
58 android.net.wifi.WifiInfo -> getSSID
59 android.net.wifi.WifiInfo -> toString
60
61 android.provider.Browser -> getAllBookmarks
62 android.provider.Browser -> getAllVisitedUrls
63 android.provider.Browser -> getVisitedHistory
64 android.provider.Browser -> getVisitedLike
65
66 android.provider.Calendar$CalendarAlerts -> query
67 android.provider.Calendar$Calendars -> query
68 android.provider.Calendar$EventDays -> query
69 android.provider.Calendar$Events -> query
70 android.provider.Calendar$Instances -> query
71
72 android.provider.CalendarContract.Attendees -> query
73 android.provider.CalendarContract.EventDays -> query
74 android.provider.CalendarContract.Reminders -> query
75
76 android.provider.CallLog$Calls -> getLastOutgoingCall
77 android.provider.Contacts$People -> loadContactPhoto
78 android.provider.Contacts$People -> openContactPhotoInputStream
79 android.provider.Contacts$People -> queryGroups
```

```
80  android.provider.ContactsContract$Contacts -> openContactPhotoInputStream
81
82  com.android.internal.telephony.CallerInfo -> getCallerId
83
84  android.provider.Telephony$Mms -> query
85  android.provider.Telephony$Sms -> query
86
87  android.telephony.TelephonyManager -> getDeviceId
88  android.telephony.TelephonyManager -> getLine1Number
89  android.telephony.TelephonyManager -> getSimSerialNumber
90  android.telephony.TelephonyManager -> getSubscriberId
91  android.telephony.TelephonyManager -> getNetworkCountryIso
92  android.telephony.TelephonyManager -> getNetworkOperator
93  android.telephony.TelephonyManager -> getNetworkOperatorName
94  android.telephony.TelephonyManager -> getSimCountryIso
95  android.telephony.TelephonyManager -> getSimOperator
96  android.telephony.TelephonyManager -> getSimOperatorName
97  android.telephony.TelephonyManager -> getSimSerialNumber
98
99  com.android.internal.telephony.IPhoneSubInfo$Stub$Proxy -> getDeviceId
100 com.android.internal.telephony.IPhoneSubInfo$Stub$Proxy -> getDeviceSvn
101 com.android.internal.telephony.IPhoneSubInfo$Stub$Proxy -> getIccSerialNumber
102 com.android.internal.telephony.IPhoneSubInfo$Stub$Proxy -> getLine1Number
103 com.android.internal.telephony.ISms$Stub$Proxy -> getAllMessagesFromIccEf
104
105 android.content.ContentResolver -> openTypedAssetFileDescriptor
106 android.content.ContentResolver -> openInputStream
107 android.content.ContentResolver -> openFileDescriptor
108 android.content.ContentResolver -> query
109
110 android.app.ContextImpl$ApplicationContentResolver -> openInputStream
111 android.app.ContextImpl$ApplicationContentResolver -> openTypedAssetFileDescriptor
112 android.app.ContextImpl$ApplicationContentResolver -> openFileDescriptor
113 android.app.ContextImpl$ApplicationContentResolver -> query
114
115 # These fields are sources
116 android.accounts.Account -> name
117 android.accounts.Account -> type
118
119 android.app.ActivityManager.RecentTaskInfo -> description
120 android.app.ActivityManager.RecentTaskInfo -> origActivity
121
122 android.app.ActivityManager.RunningAppProcessInfo -> processName
123 android.app.ActivityManager.RunningAppProcessInfo -> importanceReasonComponent
124
125 android.app.ActivityManager.RunningServiceInfo -> process
126 android.app.ActivityManager.RunningServiceInfo -> service
127
128 android.app.ActivityManager.RunningTaskInfo -> baseActivity
129 android.app.ActivityManager.RunningTaskInfo -> description
130 android.app.ActivityManager.RunningTaskInfo -> topActivity
131
132 android.net.wifi.WifiConfiguration -> SSID
133 android.net.wifi.WifiConfiguration -> BSSID
134 android.net.wifi.WifiConfiguration -> preSharedKey
135 android.net.wifi.WifiConfiguration -> wepKeys
136
137 android.net.wifi.ScanResult -> BSSID
138 android.net.wifi.ScanResult -> SSID
139
140 android.os.Build -> FINGERPRINT
```

```
141 android.os.Build −> SERIAL
142
143 # These listeners are sources, the first listed function is the function that adds the listener,
144 # the "listener:" functions are the actual listeners
145 android.accounts.AccountManager −> addOnAccountsUpdatedListener(Landroid.accounts.
        OnAccountsUpdateListener; Landroid.os.Handler; Z)
146 listener: android.accounts.OnAccountsUpdateListener −> onAccountsUpdated
147
148 android.telephony.TelephonyManager −> listen(Landroid.telephony.PhoneStateListener; I)
149 listener: android.telephony.PhoneStateListener −> onCallStateChanged
150 listener: android.telephony.PhoneStateListener −> onCellInfoChanged
151 listener: android.telephony.PhoneStateListener −> onCellLocationChanged
152
153 android.location.LocationManager −> requestLocationUpdates(J F Landroid.location.Criteria; android.
        location.LocationListener; Landroid.os.Looper;)
154 listener: android.location.LocationListener −> onLocationChanged
155
156 android.location.LocationManager −> requestLocationUpdates(Ljava.lang.String; J F Landroid.location.
        LocationListener;)
157 listener: android.location.LocationListener −> onLocationChanged
158
159 android.location.LocationManager −> requestLocationUpdates(Ljava.lang.String; J F Landroid.location.
        LocationListener; Landroid.os.Looper;)
160 listener: android.location.LocationListener −> onLocationChanged
161
162 android.location.ILocationManager$Stub$Proxy −> requestLocationUpdates(Ljava.lang.String; J F Landroid.
        location.LocationListener;)
163 listener: android.location.LocationListener −> onLocationChanged
164
165 android.location.LocationManager −> _requestLocationUpdates(Ljava.lang.String; J F Landroid.location.
        LocationListener; Landroid.os.Looper;)
166 listener: android.location.LocationListener −> onLocationChanged
167
168 android.provider.Browser −> requestAllIcons(Landroid.content.ContentResolver; Ljava.lang.String; Landroid.
        webkit.WebIconDatabase.IconListener;)
169 listener: android.webkit.WebIconDatabase$IconListener −> onReceivedIcon
170 listener: android.webkit.WebIconDatabase$IconListener −> onReceivedIcon
```

D Leak occurence tables

Table 3: Amount of times apps acquire and leak private information. a/b means: a out of b acquired pieces of information were leaked.

	account user data	account password	account name	account type	on accounts updated	query	query inputstream	Wifi MAC address	Wifi SSID	Wifi WEP keys	OS serial	OS fingerprint	subscriber id
Dropbox	1/2		0/2		0/1	5/7		1/3			0/1		
zxing						0/2	0/1		0/5	0/1			
Humble Bundle						0/1							
Tweakers						0/2							
Sudoku Free			1/2			1/6		2/2			1/1	0/2	
Reddit Sync						2/3	1/2						
Word Search						0/5							
gReader			2/6	0/1		28/177	0/2						
Shazam			0/1			19/27						2/3	1/1
Buienradar						1/2	0/2				1/3	0/1	
Twitter	7/7	1/1	1/22		0/4	4/8	3/3	1/1					
WhatsApp			5/11	6/11		38/48	7/10	1/2					0/2
iGroningen	1/1	1/1	1/1			9/9	2/2	1/1					
NS			1/4	0/1		0/1	0/1						
total	9/10	2/2	12/53	6/13	0/5	107/298	13/23	6/9	0/5	0/1	2/5	2/6	1/3

	Bluetooth address	on location changed	last known location	GSM CID	GSM LAC	IMEI	on call state changed	on cell location changed	phone number	network country ISO	network operator	sim country ISO	SIM operator	*total*
Dropbox											1/2			8/18
zxing														0/9
Humble Bundle														0/1
Tweakers		0/1	0/1								1/1			1/5
Sudoku Free		0/4	6/10			5/7			1/3	1/1	5/10	2/2		25/50
Reddit Sync						0/1					1/1			4/7
Word Search		1/1	7/9			3/3					2/4	0/1		13/23
gReader											1/1			31/187
Shazam		1/2	3/5			5/6				1/1	4/6			37/53
Buienradar		1/1	2/3	0/1	0/1	0/1					0/2			5/17
Twitter	1/1	1/1	0/1			1/1								20/50
WhatsApp		2/2				2/4	0/2	0/1	0/1	0/4	3/5	0/1	1/2	65/108
iGroningen	1/1		2/2			3/3			1/1	1/1	4/4	1/1	2/2	30/30
NS		2/2	1/5											4/14
total	2/2	8/14	21/36	0/1	0/1	19/26	0/2	0/1	2/5	3/7	22/36	3/5	3/4	243/572