Taint Analyse für Android Apps

Thomas Czogalik

Betreuer: Simon Greiner

1 Motivation

Computersysteme sind heutzutage in nahezu allen Bereichen unseres Lebens integriert. Datenlecks können deshalb fatale folgen haben. Besonders Smartphones verwalten und verarbeiten viele vertrauliche und private Daten und kommunizieren dabei oft mit der Außenwelt. In Abbildung 2 ist zu sehen, dass sich im Februar 2015 ca. 1.4 Millionen Apps im Google Play Store befanden. Diese Apps sind jedem zugänglich, der auf seinem mobilen Gerät das Betriebssystem Android installiert hat. Bei so einer großen Anzahl Apps bietet der Google Play Store eine große Angriffsfläche für Angreifer die Datenlecks ausnutzen.
Betrachten wir den folgenden Ausschnitt aus einem Java Programm in Abbildung 1. In Zeile 1 wird zunächst eine Verbindung zu einer Datenbank hergestellt. In den nächsten beiden Zeilen 2 und 3 werden ein *Statement* Objekt und ein *BufferdReader* Objekt erstellt. In Zeile 4 wird eine Benutzereingabe eingelesen und dem String *name* zugewiesen. In Zeile 5 wird der String *name* in einen vorbereiteten SQL String eingesetzt. Schließlich wird in Zeile 6 das Statement mit dem String *sql* ausgeführt.
Durch folgende Benutzereingabe: *foo; DROP TABLE users* kann die Datenbank *users* gelöscht werden. Es können aber auch mit einem beliebigen SELECT Statement Daten aus der Datenbank geholt werden.
Durch eine Überprüfung der Benutzereingabe hätte man diesen Fehler vermeiden können. Es ist aber nicht immer so einfach Datenlecks im Code zu finde. Deshalb wäre es hilfreich, wenn man den Fluss sensitiver Daten nachvollziehen könnte. Dies ist mithilfe einer Taint Analyse möglich.

```
1 connection = ...
2 stmt = connection.createStatement();
3 BufferedReader br = ...
4 String name = br.readLine();
5 String sql = "SELECT * FROM users WHERE name=" + name + ";";
6 stmt.executeQuery(sql);
```

Abb. 1. Beispiel bei dem SQL Injections möglich sind

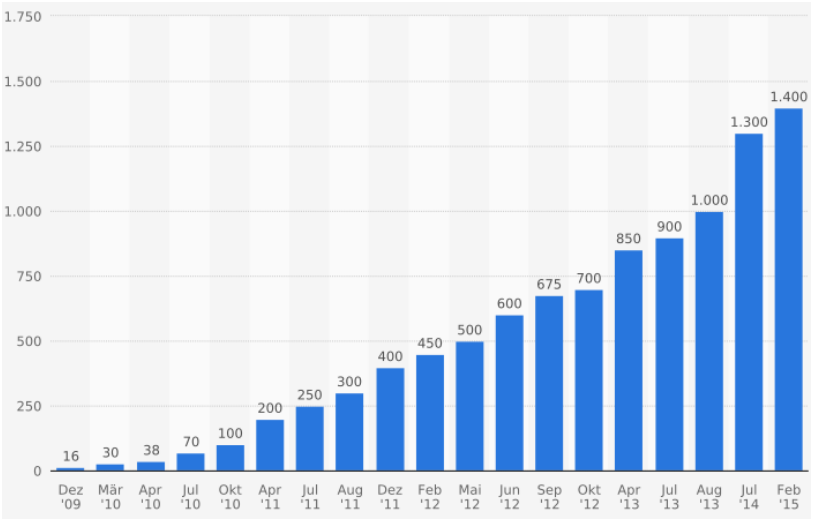

Abb. 2. Anzahl der verfügbaren Apps im Google Play Store von Dezember 2009 bis Februar 2015 (in 1.000), aus [6]

Im folgenden Abschnitt 2 wird zunächst auf die Grundlagen der Taint Analyse eingegangen um sie dann anschließend im Abschnitt 3 formal darzustellen. Außerdem werden in Abschnitt 5 Eigenschaften für die Taint Analyse vorgestellt, die zur einer besseren Präzision führen. Im Anschluss wird in Abschnitt 6 auf die Probleme bei Android eingegangen die bei einer Taint Analyse auftreten. Schließlich wird in Abschnitt 7 FlowDroid vorgestellt, dass eine statische Taint Analyse für Android Apps ist.

2 Taint Analyse

Die Taint Analyse ist eine Datenflussanalyse. Man unterscheidet zwischen statischer und dynamischer Taint Analyse. Bei einer statischen Analyse wird der Quelltext einer Reihe formaler Prüfungen unterzogen. Der Vorteil dabei ist, dass das Kompilat nicht ausgeführt werden muss, im Gegensatz zur dynamischen Analyse, die ein laufendes Programm voraussetzt. Dies ist hilfreich, da heutige Malware erkennen kann ob sie überwacht wird und ihr Verhalten der Situation anpasst.
Im folgenden wird sich auf die statische Taint Analyse beschränkt.

Die Taint Analyse nimmt zunächst an, dass jede von außen veränderbare Variable ein Sicherheitsrisiko birgt. Ihr Ziel ist es, die Software gegen externe Angriffe, sowie interne Risiken abzusichern. Dazu sucht die Taint Analyse nach Datenflüssen von möglichen tainted Sources zu einem Sink. Als Source wird eine Funktion bezeichnet, die Quelle sensitiver Daten ist. Ein Sink ist eine Funktion, die

Daten möglicherweise an nicht vertrauenswürdige Beobachter weitergibt. Welche Funktionen im einzelnen Sources und Sinks sind, muss vor dem ausführen der Taint Analyse spezifiziert werden. In dem Beispiel aus Abbildung 1 wäre die Funktion *readLine()* eine Source, da Daten von außen eingeführt werden (Benutzereingabe) und die Funktion *executeQuery(sql : String)* ein Sink, da hier möglicherweise sensitive Daten weitergegeben werden.

$$\frac{src(m,l) \qquad m \to o}{tainted(o,l)} \ (SOURCE)$$

$$\frac{tainted(o_1,l) \qquad m \to o_1 \qquad n \to o_2 \qquad transfer(m,n)}{tainted(o_2,l)} \ (TRANSFER)$$

$$\frac{tainted(o,so) \qquad m \to o \qquad sink(m,si)}{flow(so,si)} \ (SOURCE)$$

Abb. 3. Schlussregeln für die Taint Analyse

3 Formal

Im Folgendem wird die Taint Analyse mithilfe von Schlussregeln in Abbildung 3 formalisiert. Eine Schlussregel besteht aus einer Menge von Prämissen, die über dem Schlussstrich stehen und einer Konklusion die darunter steht. Wenn die Prämissen alle wahr sind folgt die Konklusion.
Die Regeln setzten sich aus den folgenden Prämissen zusammen:

Prämisse	**Bedeutung**
src(m,so)	Dem Parameter m werden tainted Daten aus einer Source mit dem Namen *so* zugewiesen
sink(m,si)	Der Parameter m wird in einen Sink mit dem Namen *si* übergeben
tainted(o,so)	Der Objektrepräsentant o wird durch eine Source mit dem Namen *so* tainted
flow(so,si)	Es gibt einen Datenfluss von einer Source mit dem Namen *so* zu einem Sink mit Namen *si*
transfer(m,n)	Es werden Daten von Parameter m zu Parameter n übertragen
$m \to o$	Die Variable m zeigt auf einen Objektrepräsentanten o

Die Namen der Sources und Sinks stammen jeweils aus einem Source-Label-Set, bzw. einem Sink-Label-Set. Diese Sets enthalten alle Sources und Sinks die

für die jeweilige Taint Analyse spezifiziert wurden. Als Objektrepräsentant wird ein Objekt im Sinne der Analyse bezeichnet.
Die erste Regel ist die SOURCE Regel. Sie beschreibt die Zuweisung von tainted Daten durch eine Source. Die SOURCE Regel setzt sich aus den Prämissen *src(m,l)* und $m \rightarrow o$ zusammen. Wenn diese beiden Prämissen wahr sind folgt die Konklusion *tainted(o,l)*.
Möchte man die SOURCE Regel auf das Beispiel aus Abbildung 1 anwenden, muss man zunächst die Sources und Sinks definieren und diese in das jeweilige Set eintragen. In das Source-Label-Set wird die Funktion *readline* aus Zeile 4 eingetragen, da hier durch die Benutzereingabe potentiell gefährliche Daten reinkommen können. In das Sink-Label-Set wird die Funktion *executeQuery* eingetragen, da hier möglicherweise tainted Daten weitergegeben werden können. Die SOURCE Regel wird in Zeile 4 den String *name* als tainted durch die Source *readLine* markieren, da die beiden Prämissen wahr sind.
Die zweite Regel TRANSFER beschreibt die Übertragung von tainted Daten. Sie besteht aus den Prämissen *tainted*(o_1,*l)*, $m \rightarrow o_1$, $n \rightarrow o_2$ und *transfer(m,n)*. Wenn alle Prämissen wahr sind, dann folgt die Konklusion *tainted*(o_2,*l)*.
Die TRANSFER Regel wir in dem Beispiel aus Abbildung 1 in Zeile 5 angewendet, da hier Daten vom String *name* zum String *sql* übertragen werden. Da auch hier alle Prämissen der TRANSFER Regel erfüllt sind, wird der String *sql* auch als tainted markiert.
Die letzte Regel SOURCE beschreibt den Datenfluss von tainted Daten zu einem Sink. Sie besteht aus den Prämissen *tainted(o,so)*, $m \rightarrow o$ und *sink(m,si)*. Sind alle Prämissen wahr, dann folgt die Konklusion *flow(so,si)* und der Sink *si* wird als tainted markiert. Die SOURCE Regel wird in dem Beispiel aus Abbildung 1 in Zeile 6 angewendet, da hier der String *sql* dem Sink *executeQuery* übergeben wird. Auch hier sind alle Prämissen der SOURCE Regel erfüllt und somit folgt die Konklusion und der Sink *executeQuery* wird als tainted markiert.
Mit diesen drei Regeln und Fixpunktiteration kann man eine Taint Analyse durchführen.

4 Sanitization

Mithilfe von Sanitization Funktionen lassen sich Code Injections verhindern. Als Code Injection bezeichnet man das ausnutzen eines Programmfehlers indem man Code in das Programm einschleust.
In Abbildung 1 würde zum Beispiel die Funktion *onlyLetters(s : String) : String* die Benutzereingabe als Parameter entgegen nehmen, alle nicht Buchstaben aus der Eingabe entfernen und einen String zurückgeben, der nur aus Buchstaben besteht und somit eine Code Injection verhindern. Einige Programmiersprachen bieten solche Säuberungsfunktionen an. PHP hat zum Beispiel die Funktion *htmlentities*. Diese Funktion konvertiert Zeichen, die in HTML besondere Bedeutung haben in ihre HTML Instanzen. Aus dem Zeichen '<' wird '<'.
Eine Taint Analyse die Sanitization unterstützt, hat den Vorteil, dass gesäuberte

Daten nicht länger als tainted angesehen werden und nicht mehr als tainted in der Analyse verbreitet werden.

5 Präzision

Eine gute Analyse sollte in der Lage sein keine false-positives zu finden. Dadurch wird zum einen die manuelle Aussortierung dieser durch den Benutzer vermieden und zum anderen die Präzision der Analyse erhöht. Dabei ist ein false-positive ein Ergebnis das anzeigt, dass eine bestimmte Voraussetzung erfüllt ist, obwohl dies nicht der Fall ist. Eine Analyse die eine oder mehrere der folgenden Eigenschaften implementiert hat den Vorteil, dass weniger false-positives gefunden werden.

5.1 Fluss-Sensitivität

Eine Analyse die Fluss-Sensitiv ist, beachtet die Reihenfolge der Programmbefehle. In Abbildung 4 wird dem String *s* in Zeile 2 zunächst ein harmloser Wert zugewiesen. Danach wird dieser String in Zeile 3 einem Sink übergeben. Im Anschluss werden in Zeile 4 dem String *s* tainted Daten aus einem Source zugewiesen.
Eine Fluss-Sensitive Analyse wird den Sink in Zeile 3 nicht als tainted markieren, da sie die Reihenfolge der Programmbefehle beachtet.

```
1 void flowSensitive() {
2   String s = "value";
3   sink(s);
4   s = source();
5 }
```

Abb. 4. Fluss-Sensitivität

5.2 Kontext-Sensitivität

Ist eine Analyse Kontext-Sensitiv, beachtet sie den Kontext des Aufrufs. Dies bedeutet, dass eine solche Analyse zu dem Aufrufer einer Funktion zurück springen kann und somit den genauen Funktionsaufrufer bestimmen kann, während eine Analyse die nicht Kontext-Sensitiv ist alle möglichen Aufrufer berücksichtigen muss. In Abbildung 5 werden den beiden Strings *s1* und *s2* die Rückgabe der Funktion *id(s : String) : String* zugewiesen. Die Funktion *id* ist dabei die Identitätsfunktion. Nun wird *id* einmal in Zeile 2 mit tainted Daten ausgeführt und *s1* zugewiesen und noch einmal in Zeile 3 mit einem harmlosen String und *s2* zugewiesen. Schließlich wird *s2* dem sink in Zeile 4 übergeben. Eine Kontext-Sensitive Analyse kann hier zwischen den beiden Aufrufen in Zeile 2 und 3 unterscheiden und wird den Sink in Zeile 4 nicht als tainited markieren.

```
1 void contextSensitive() {
2    String s1 = id(source());
3    String s2 = id("123");
4    sink(s2);
5 }
6
7 String id(String s) {
8    return s;
9 }
```

Abb. 5. Kontext-Sensitivität

5.3 Objekt-Sensitivität

Die Objekt-Sensitivität ist ein Subtyp der Kontext-Sensitivität. Eine Objekt-Sensitive Analyse beachtet nämlich den Kontext des aufrufenden Objekts. Das heißt, dass zu dem Objekt zurück gesprungen werden kann, dass eine Methode aufruft. In Abbildung 6 werden zunächst zwei *Foo* Objekte *o1* und *o2* erstellt. Die Klasse *Foo* besteht aus einem String mit Namen *value* und einer *getValue* Methode, die *value* zurückgibt. In Zeile 4 und 5 wird *o1* ein harmloser String übergeben und *o2* tainted Daten aus einer Source. Im Anschluss wird in Zeile 7 die Funktion *getValue* auf *o1* aufgerufen und der Rückgabewert einem Sink übergeben. Eine Objekt-Sensitive Analyse kann zwischen den beiden Objekten *o1* und *o2* unterscheiden und markiert den Sink in Zeile 7 nicht als tainted.

```
1  void objectSensitive() {
2     Foo o1 = new Foo();
3     Foo o2 = new Foo();
4     o1.value = "123";
5     o2.value = source();
6
7     sink(o1.getValue());
8  }
9
10 class Foo {
11    String value;
12
13    String getValue() {
14       return value;
15    }
16 }
```

Abb. 6. Objekt-Sensitivität

5.4 Feld-Sensitivität

Eine Feld-Sensitive Analyse kann Felder einer Klasse einzeln betrachten. Macht eine Analyse dies nicht, werden die Felder zu ihrem Basis Objekt zurückgeführt. In Abbildung 7 wird in Zeile 2 zunächst ein *Foo* Objekt *o* erstellt. Die Klasse *Foo* enthält hier 2 Felder *field1* und *field2*. In Zeile 3 und 4 wird *field1* ein harmloser String zugewiesen während *field2* tainted Daten zugewiesen werden. Schließlich wird *field1* in einen Sink in Zeile 6 übergeben. Bei einer Feld-Sensitiven Analyse wird hier der Sink nicht als tainted markiert.

```
1  void fieldSensitive() {
2     Foo o = new Foo();
3     o.field1 = "123";
4     o.field2 = source();
5
6     sink(o.field1);
7  }
8
9  class Foo {
10    String field1;
11    String field2;
12 }
```

Abb. 7. Feld-Sensitivität

6 Android

Android ist sowohl ein Betriebssystem als auch eine Software-Plattform für mobile Geräte. Es bietet Entwicklern eine Schnittstelle an, um auf verschiedene Systemfunktionalitäten des mobilen Gerätes zuzugreifen. Der Nutzerstandort ist zum Beispiel eine solche Funktionalität. Der Entwickler muss die von ihm genutzte Funktionalität in die *AndroidManifest.xml* eintragen. Der Benutzer, der die App installieren möchte muss dieser die Berechtigung geben auf diese Funktionalitäten zugreifen zu dürfen. Oft werden zu viele Berechtigungen gefordert. Dies geschieht entweder aus Unkenntnis oder mit bösen Absichten. Außerdem sind viele Berechtigungen zu mächtig und können ausgenutzt werden. Eine socialnetwork App zum Beispiel, möchte die Berechtigung *android.permission.READ_CONTACTS* um neue Freunde anhand der E-Mail Adresse vorschlagen zu können. Sie kann aber mit dieser Berechtigung auch Telefonnummern und sonstige Kontaktdaten auslesen, auch wenn der Kontakt nicht bei dem Dienst angemeldet ist. Dies kann ausgenutzt werden.
Um dem entgegenzuwirken kann man die Datenflüsse einer Android App mithilfe der Taint Analyse analysieren.

Eine Android App kann aus vier Arten von Komponenten bestehen.

Activity	Einzelner Screen, der für den Benutzer sichtbar ist
Service	Eine Aktion die im Hintergrund abläuft
Content Provider	Ist zuständig für das Lesen und Speichern von Daten
Broadcast Reciever	Wartet auf ein globales Ereignis und führt eine vordefinierte Aktion aus

Da eine Android App aus einer beliebigen Zusammensetzung dieser Komponenten bestehen kann, zum Beispiel drei Activities und einem Service, entstehen Schwierigkeiten für eine Taint Analyse. Zunächst hat eine Android App keine zentrale *main* Methode, sondern kann aufgrund der verschiedenen Komponenten mehrere Einstiegspunkte haben. Es gibt zwar eine Haupt-Activity, aber welche Komponente als nächstes an die Reihe kommt ist im allgemeinen nicht bestimmbar, da diese von der Benutzereingabe abhängig sind. Außerdem haben die einzelnen Komponenten besonderes Verhalten. Sie können zum Beispiel gestartet und beendet werden oder angehalten werden, wenn der Speicher voll ist und fortgesetzt, wenn wieder Speicher frei ist. Somit entsteht ein komplizierter Lebenszyklus, mit dem die Taint Analyse umgehen muss, da tainted Daten zwischen den Komponenten hin und her gereicht werden können in beliebiger Reihenfolge und mit verschiedenem Verhalten der einzelnen Komponenten.

7 FlowDroid

FlowDroid[1] ist eine statische Taint Analyse. Sie ist Fluss-, Kontext-, Objekt- und Feld-Sensitiv. Zur Bestimmung der Sources und Sinks benutzt FlowDroid das Tool SuSi. SuSi ist ein machine-learning Tool, dass vollautomatisch den Android Source Code analysiert und eine Liste von Sinks und Sources generiert. Außerdem generiert FlowDroid vor der Taint Analyse einen präzisen Android Lebenszyklus.
Aufgrund der vielen möglichen Einstiegspunkte bei einer Android App, wird bei FlowDroid zunächst eine dummy-main-Methode erstellt. Diese Methode wird individuell für jede App erstellt. Die dummy-main enthält nur Teile des Lebenszyklus die auch auftreten können. Um diese zu identifizieren werden die XML Konfigurationsdateien und der Quellcode analysiert. Danach ruft die dummy-main-Methode nacheinander Teile des Zyklus auf. In Abbildung 9 ist eine Klasse *LeakageApp*, die von *Activity* erbt und die Methode *onRestart* überschreibt. Außerdem ist die methode *sendMessage* im Layout-XML definiert und in Abbildung 9 implementiert. In Abbildung 8 ist die von FlowDroid generierte dummy-main-Methode zu sehen. Es werden nur die Teile des Lebenszyklus verwendet, die in dem Beispiel vorkommen, wie die Methoden *onRestart* und *sendMessage*,

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

bzw. vom Betriebssystem für den Lebenszyklus der App gebraucht werden, wie *onCreate*, *onStart*, *onResume*, *onPause*, *onStop* und *onDestroy*.

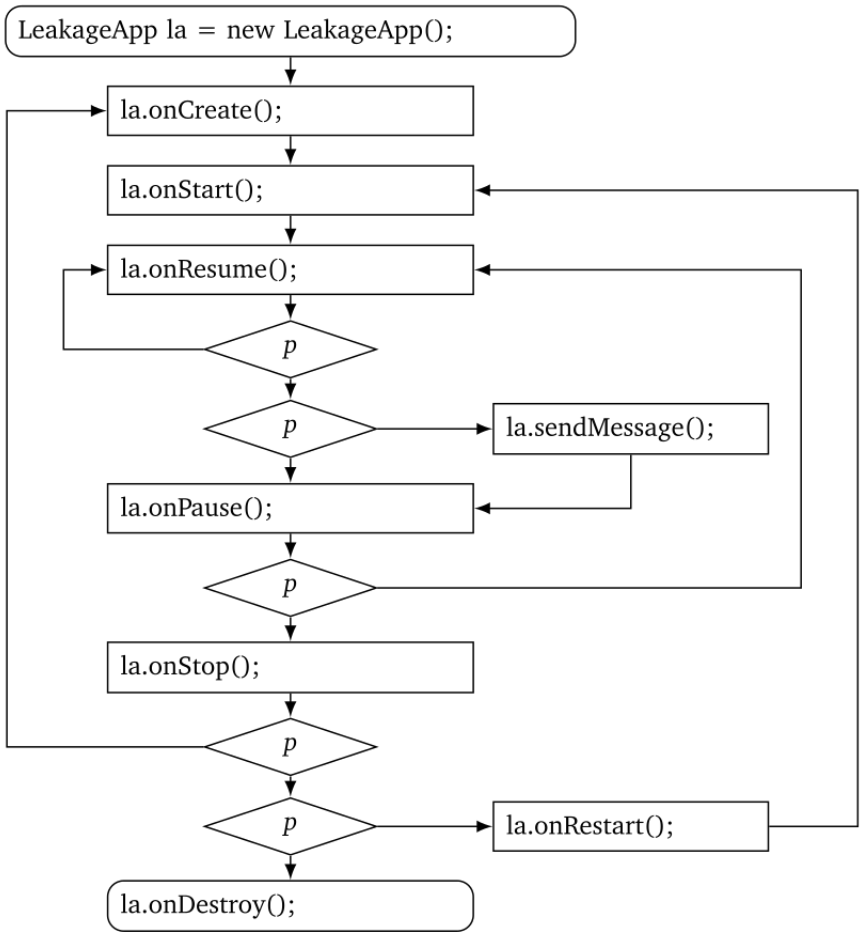

Abb. 8. Aktivitätsdiagramm für die dummy Methode aus Abbildung 9

```
public class LeakageApp extends Activity {
   protected void onRestart() {
      ...
   }
   //Callback Methode. In Layout-XML definiert
   public void sendMessage(View v) {
      ...
   }
}
```

Abb. 9. Lebenszyklus Code Beispiel

8 Evaluation

Für die Evaluierung von FlowDroid wurde die Testumgebung DroidBench [2] verwenden. DroidBench enthielt zum damaligen Zeitpunkt 39 kleine Test Apps. FlowDroid wurde mit den beiden komerziellen Analyse Tools *App Scan* von IBM und *Fortify SCA* von HP verglichen.
Es gab dabei folgende Test Kategorien:

Kategorie	**Beschreibung**
Arrays and Lists	Umgang mit Arrays und Listen
Callbacks	Umgang mit Callbacks, sowohl in XML definiert als auch im Code
Field and Object Sensitivity	Feld- und Objekt-Sensitive Tests
Inter-App Communication	Umgang mit tainted Daten, die einer anderen Activity übergeben werden
Lifecycle	Umgang mit Methoden des Lebenszyklus
General Java	Allgemeine Java Tests
Miscellaneous Android Specific	Verschiedene Android spezifische Tests wie, Passwort Feld Eingabe wird geleakt

In dem Evaluierungsbogen in Abbildung 10 ist zu erkennen, dass FlowDroid in den Kategorien *Callbacks* und *Lifecycle* deutlich besser ist, als *Fortify* und *App Scan*. Dies ist auf die präzise Modelierung des Lebenszyklus zurückzuführen. FlowDroid findet aber auch nicht alle Fehler. In dem Test *IntentSink1* findet FlowDroid den Fehler nicht, da in dem Test tainted Daten aus einer Komponente in die nächste übergeben werden und diese Komponente die Daten direkt wieder zurück schickt. FlowDroid verfolgt tainted Daten nur innerhalb von Komponenten. Deshalb findet FlowDroid den Fehler in diesem Test nicht.

[2] `https://github.com/secure-software-engineering/DroidBench`

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan Source	Fortify SCA	FlowDroid
Arrays and Lists			
ArrayAccess1			⋆
ArrayAccess2	⋆	⋆	⋆
ListAccess1	⋆	⋆	⋆
Callbacks			
AnonymousClass1	○	⊛	⊛
Button1	○	⊛	⊛
Button2	⊛ ○ ○	⊛ ○ ○	⊛ ⊛ ⊛ ⋆
LocationLeak1	○ ○	○ ○	⊛ ⊛
LocationLeak2	○ ○	○ ○	⊛ ⊛
MethodOverride1	⊛	⊛	⊛
Field and Object Sensitivity			
FieldSensitivity1			
FieldSensitivity2			
FieldSensitivity3	⊛	⊛	⊛
FieldSensitivity4	⋆		
InheritedObjects1	⊛	⊛	⊛
ObjectSensitivity1			
ObjectSensitivity2	⋆		
Inter-App Communication			
IntentSink1	⊛	⊛	○
IntentSink2	⊛	⊛	⊛
ActivityCommunication1	⊛	⊛	⊛
Lifecycle			
BroadcastReceiverLifecycle1	⊛	⊛	⊛
ActivityLifecycle1	⊛	⊛	⊛
ActivityLifecycle2	○	⊛	⊛
ActivityLifecycle3	○	○	⊛
ActivityLifecycle4	○	⊛	⊛
ServiceLifecycle1	○	○	⊛
General Java			
Loop1	⊛	○	⊛
Loop2	⊛	○	⊛
SourceCodeSpecific1	⊛	⊛	⊛
StaticInitialization1	○	⊛	○
UnreachableCode		⋆	
Miscellaneous Android-Specific			
PrivateDataLeak1	○	○	⊛
PrivateDataLeak2	⊛	⊛	⊛
DirectLeak1	⊛	⊛	⊛
InactiveActivity	⋆	⋆	
LogNoLeak			
Sum, Precision and Recall			
⊛, higher is better	14	17	26
⋆, lower is better	5	4	4
○, lower is better	14	11	2
Precision $p = ⊛/(⊛ + ⋆)$	74%	81%	86%
Recall $r = ⊛/(⊛ + ○)$	50%	61%	93%
F-measure $2pr/(p+r)$	0.60	0.70	0.89

Abb. 10. DroidBench Evaluierung, aus [5]

Auch in dem Test *StaticInitialization1* findet FlowDroid den Fehler nicht. Hier liegt der Grund in dem von FlowDroid generierten Call-Graphen. Bei der Generierung geht FlowDroid davon aus, dass die komplette statische Initialisierung zu beginn stattfindet, was in diesem Testfall nicht der Fall ist.
Im Ergebnis ist FlowDroid aber deutlich besser als *Fortify* und *App Scan*. FlowDroid erzielt eine Präzision von 86%, während *Fortify* 81% und *App Scan* 74% erreichen. Als Präzision wird hier das Verhältnis zwischen gefunden Fehlern und false-positives bezeichnet. Außerdem findet FlowDroid 93% aller Fehler in den Testfällen während *Fortify* 61% und *App Scan* nur 50% aller Fehler finden.
Mit FlowDroid wurden zusätzlich die damals Top 500 Apps des Google Play Stores getestet. Dabei wurden in den meisten Apps ein Datenleck gefunden, die aber keine bösartigen Absichten erkennen liesen.
Außerdem wurden 1000 Malware Apps des VirusShare Projects analysiert. Es wurden pro App ca. zwei Datenlecks gefunden.
Das zeigt, dass FlowDroid sowohl Datenlecks finden kann die unabsichtlich entstehen, als auch solche die absichtlich in eine App eingebaut werden.

9 Ausblick

FlowDroid macht vieles richtig, aber wie die Evaluierung gezeigt hat, findet FlowDroid noch nicht alle möglichen Datenlecks.
Zum Beispiel verfolgt FlowDroid tainted Daten nur innerhalb einer Komponente. Es sollte aber auch möglich sein, tainted Daten zwischen den Komponenten zu verfolgen. Dazu könnte ein Sink der einen Komponente gleichzeitig die Source der anderen Komponenten sein.
Außerdem hat FlowDroid keine Sanitization implementiert. Dies ist jedoch notwendig, damit der Programmierer auf gefundene Datenlecks reagieren und tainted Daten auch säubern kann.
Für die Zukunft wäre es auch wichtig, dass FlowDroid unterscheiden kann zwischen notwendigen und bösartigen Datenlecks. Eine Navigationsapp benötigt den Benutzerstandort. Eine Taschenlampen App jedoch nicht.
Schließlich wäre es auch hilfreich, wenn die Analyse für bestimmte Arten von Sinks und Sources einschränkbar wäre. Der Vorteil hierbei wäre, dass man sich bestimmte Datenflüsse näher anschauen könnte.

10 Ergebnis

In dieser Seminararbeit wurde die statische Taint Analyse vorgestellt und erklärt. Die Taint Analyse wurde mithilfe von Schlussregeln formalisiert und es wurden Analyseeigenschaften vorgestellt, mithilfe derer man eine Taint Analyse präziser machen kann. Außerdem wurde mit Sanitization ein Mechanismus vorgestellt mit dem man tainted Daten säubern kann und diese somit von der Taint Analyse nicht mehr als tainted weiterverbreitet werden.
Daraufhin wurde FlowDroid vorgestellt, eine statische Taint Analyse für Android Apps. Es wurde dargestellt wie FlowDroid mit den Schwierigkeiten umgeht, die

bei einer Taint Analyse für Android Apps vorkommen. Außerdem wurde FlowDroid mit den beiden kommerziellen Tools *Fortify* von HP und *App Scan* von IBM verglichen und hat gezeigt, dass es deutlich besser beim auffinden aller Datenlecks ist.
Schließlich wurden auch die Beschränkungen von FlowDroid aufgezeigt und mögliche Verbesserungen für die Zukunft der Analyse vorgestellt.

Literatur

1. Steven Arzt, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yvesle Traon, Damien Octeau, Patrick McDaniel, and Siegfried Rasthofer. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *PLDI 2014*. `http://www.bodden.de/pubs/far+14flowdroid.pdf`.
2. Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. *2008 IEEE Symposium on Security and Privacy*, pages 387 – 401.
3. Yu Feng, Saswat Anand, Alex Aiken, and Isil Dillig. Apposcopy: Semantics-based detection of android malware through static analysis. *FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576 – 587.
4. Chrisitan Fritz. Flowdroid: A precise and scalable data flow analysis for android, Juli 2013. `https://ec-spride.sit.fraunhofer.de/fileadmin/user_upload/Group_EC_Spride/FinishedTheses/Masterthesis20130702.pdf`.
5. Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for android applications. *EC SPRIDE Technical Report TUD-CS-2013-0113*, Mai 2013. `https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_CASED/Publikationen/TUD-CS-2013-0113.pdf`.
6. Business Insider. Anzahl der verfügbaren apps im google play store von dezember 2009 bis februar 2015 (in 1.000). Statista. `http://de.statista.com/statistik/daten/studie/74368/umfrage/anzahl-der-verfuegbaren-apps-im-google-play-store/`, [Stand 24.03.2016].
7. Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. `https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_EC-Spride/Publikationen/SuSi2014.pdf`.