Taint Analyse für Android Apps

Thomas Czogalik

Betreuer: Simon Greiner

1 Motivation

Durch fehlerhafte oder absichtlich manipulierte Programme können Datenlecks enstehen. Diese Datenlecks können ausgenutzt werden. In Abbildung 1 ist ein Ausschnit aus einem Java Programm. Es wird zunächst eine Verbindung zu einer Datenbank hergestellt. Im Anschluß soll ein SQL Statement ausgeführt werden. Das SQL Statement in Zeile sechs hängt von einer Benutzereingabe aus Zeile fünf ab. Ein Angreifer könnte in diesem Fall durch folgende Eingabe: "*foo; DROP TABLE users*" die Datenbank *users* löschen. Er könnte aber auch mit einem SELECT Statement Daten aus der Datenbank holen.
So ein Fehler lässt sich durch Überprüfung der Benutzereingabe vermeiden. Es ist aber nicht immer so einfach einen solchen zu erkennen.
Da Computersysteme heutzutage in nahezu allen Bereichen unseres Lebens intigriert sind, können solche Datenlecks fatale folgen haben. Besonders Smartphones verwalten und verarbeiten viele vertauliche und private Daten und komunizieren dabei oft mit der Außenwelt. Im Februar 2015 befanden sich im Google Play Store ca. 1.4 Millionen Apps. 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 solche Datenlecks ausnutzen. Deshalb ist es notwendig, den Fluss sensitiver Daten nachvollziehen zu können. Dies ist mithilfe einer Taint Analyse möglich.
Der Autor möchte im folgenden zunächst auf die Grundlagen der Taint Analyse eingehen und danach die Probleme vorstellen, die bei einer Taint Analyse für Android Apps auftreten. Im Anschluß wird FlowDroid vorgestellt eine Taint Analyse für Android Apps.

```
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);
```

Fig. 1. Beispiel bei dem SQL Injections möglich sind

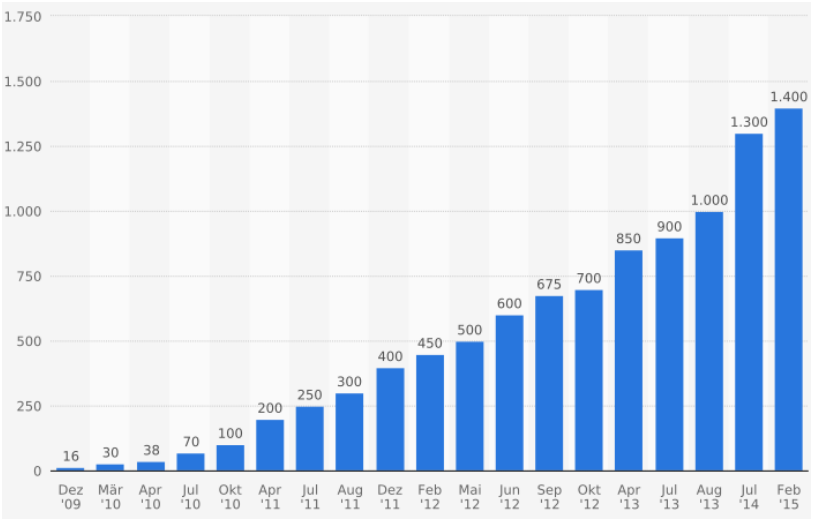

Fig. 2. Anzahl der verfügbaren Apps im Google Play Store von Dezember 2009 bis Februar 2015 (in 1.000), aus [6]

2 Taint Analyse

2.1 Statische Taint Analyse

Die Taint Analyse ist eine Datenflussanalyse. Bei der Tain Analyse unterscheidet man zwischen statischer und dynamischer Taint Analyse. Der Vorteil der statischen Variante ist, dass das Kompilat nicht ausgeführt werden muss. 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.

2.2 Spezifikation

Die Idee der Taint Analyse ist, 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 der Taint Analyse mitgegeben werden.

2.3 Formal

Im Folgendem wird die Taint Analyse mithilfe von Schlussregeln formalisiert(3). Die SOURCE Regel beschreibt die Einfuhr von tainted Daten durch einen Source.

src(m,l) bedeutet, dass dem Parameter m Daten aus einer Source zugewiesen werden mit dem Namen l. l ist der Name einer Source, die aus einem Source-Label-Set stammt. Wenn m auf einen Objektrepräsentanten o zeigt, ist o durch die Source l tainted.
Als Objektrepräsentant wird ein Objekt im Sinne der Analyse bezeichnet. Da nicht sicher ist auf wieviele Objekte tatsächlich zur Laufzeit ein Zeiger zeigt, fasst man diese Menge zusammen und wählt einen Objekträpresentanten.
Die zweite Regel TRANSFER beschreibt die Übertragung von tainted Daten. *transfer(m,n)* bedeutet, dass Daten vom Parameter m zu n übertragen werden. Wenn außerdem der Objektrepräsentant *o1* von einer Source l tainted ist und m auf *o1* und n auf ein *o2* zeigt, dann ist auch *o2* durch l tainted.
Die letzte Regel SOURCE beschreibt den Datenfluss von verschmutzen Daten zu einem Sink. Wenn der Objektrepräsentant o durch eine Source *so* tainted ist und die Variable m, die auf o zeigt einem Sink mit dem Namen *si* übergeben wird, dann gibt es einen Datenfluss von der Source *so* zum Sink *si*. *si* ist dabei der Name eines Sinks aus einem Sink-Label-Set.
Mit diesen drei Regeln kann man eine Taint Analyse durchführen indem man diese solange immer wieder ausführt, bis sich am Ergebniss nichts mehr ändert.

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

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

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

Fig. 3. Schlussregeln für die Taint Analyse

2.3.1 Beispiel Im folgenden wird dargestellt wie die Anwendung der Regeln auf das Beispiel aus Abbildung 1 aussieht.
Zunächst müssen die Sources und Sinks definiert werden. Dazu wird die Funktion *readLine* aus Zeile vier in das Source-Label-Set eingefügt. In das Sink-Label-Set wird die Funktion *executeQuery* aus Zeile sechs eingefügt. Die Taint Analyse wird zunächst die SOURCE Regel in Zeile vier anwenden, da dort Daten aus einer Source dem String *name* zugewiesen werden. Somit wird der String *name* als tainted markiert. Als nächstes wird die TRANSFER Regel in Zeile fünf angewendet. Hier werden Daten vom String "name" zum String *sql* übertragen. Deshalb wird auch hier der String *sql* als tainted markiert. Schließlich wird in

Zeile sechs die SINK Regel angewendet. Hier wird der String *sql* in den Sink *executeQuery* übergeben. Da *sql* tainted ist, wird der Sink *executeQuery* auch als tainted markiert. Somit wurde ein tainted Datenfluss von der Source *readLine* zum Sink *executeQuery* gefunden.

2.4 Sanitization

Nachdem die Taint Analyse Datenflüsse von sources zu sinks entdeckt hat, sollte dem Programmierer die Möglichkeit gegeben werden, seine Daten zu überprüfen und zu säubern.
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. 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 "<".
Neben den eingebauten sanitization Funktionen können auch eigene Funktionen geschrieben werden. Dies kann jedoch gefährlich werden, da Daten die einen Säuberungsprozess durchlaufen haben als harmlos angesehen werden. Das kann zu Problemen führen, wenn die Funktion nicht vollständig oder falsch ist. Grund für solche Fehler ist, dass es schwierig ist im Vornherein festzulegen welche Fälle durch eine Sanitization Funktion beachtet werden müssen. Außerdem müssen die Sanitization und die Sink Funktion zusammen passen. In Abbildung 1 würde die Funktion *htmlentities* die Eingabe nicht säubern.

2.5 Präzision

Eine gute Analyse sollte in der Lage sein false-positives nicht zu melden. Dadurch wird die manuelle Aussortierung dieser durch den Benutzer vermieden. Als false-positives werden Fehlermeldungen bezeichnet, die einen Fehler anzeigen, obwohl keiner vorliegt. Mit den folgenden Eigenschaften lassen sich einige false-positives vermeiden.

2.5.1 Fluss-Sensitivität In Abbildung 4 wird einem String *s* in Zeile zwei zunächst ein harmloser Wert zugewiesen. Im Anschluss wird dieser String einem Sink übergeben. Obwohl die tainted Daten, die dem String *s* in Zeile vier zugewiesen werden nie in den Sink kommen, wird der Sink trotzdem als tainted markiert. Grund dafür ist, dass die Reihenfolge hier nicht beachtet wird. Beachtet man diese meldet die Analyse hier keinen Fehler.
Diese Eigenschaft nennt man Fluss-Sensitivität.

2.5.2 Kontext-Sensitivität Obwohl *s2* in Abbildung 5 keine tainted Daten zugewiesen bekommt, wird der Sink in Zeile vier als tainted markiert. Der Grund hierfür ist die Funktion *id(s : String) : String* aus Zeile sieben. Da *s1* und *s2* die

Funktion *id* aufrufen, zeigt das *s* aus Zeile acht sowohl auf *s1* als auf *s2*. Da *id* mit den tainted Daten aus dem Source aufgerufen wird, werden *s1* und *s2* als tainted markiert.
Beachtet man jedoch den Kontext des Aufrufers, zeigt das *s* aus Zeile acht entweder auf *s1* oder auf *s2*. Somit wird bei dieser herangehensweise kein Fehler gemeldet.

2.5.3 Objekt-Sensitivität Ein ähnliches Problem wie zuvor ist in Abbildung 6 zu sehen. Die Funktion *getValue() : String* der Klasse Foo zeigt sowohl auf *o1* als auf *o2*. Obwohl der Sink in Zeile sieben keine tainted Daten erhält wird hier ein Fehler gemeldet, denn die Analyse kann nicht zwischen den einzelnen Objekten unterschieden.
Eine Analyse die Objekt-Sensitiv ist meldet hier keinen Fehler.
Die Objekt-Sensitivität ist ein Subtyp der Kontext-Sensitivität.

2.5.4 Feld-Sensitivität In der letzten Abbildung (7) ist ein Objekt *o*, dass mehrere Felder besitzt. Einem der Felder werden tainted Daten aus einer Source zugewiesen. Dem anderen Feld werden harmlose Daten zugewiesen. Obwohl das harmlose Feld dem Sink übergeben wird, wird der Sink trotzdem als tainted markiert. Der Grund hierfür ist, dass Felder bei Feld-Insensitiven Analysen zu ihrem Basis Objekt zusammengeführt werden und als ganzes betrachtet werden. Da hier *field2* tainted ist, wird das ganze Objekt *o* als tainted markiert und somit auch das *field1*.
Bei einer Feld-Sensitiven Analyse werden die Felder einzeln betrachtet und kein Fehler wird gemeldet.

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

Fig. 4. Fluss-Sensitivität

3 Android

Android ist sowohl ein Betriebssystem als auch eine Software-Platform 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,

```
void contextSensitive() {
   String s1 = id(source());
   String s2 = id("123");
   sink(s2);
}

String id(String s) {
   return s;
}
```

Fig. 5. Kontext-Sensitivität

```
void objectSensitive() {
   Foo o1 = new Foo();
   Foo o2 = new Foo();
   o1.value = "123";
   o2.value = source();

   sink(o1.getValue());
}

class Foo {
   String value;

   String getValue() {
      return value;
   }
}
```

Fig. 6. Objekt-Sensitivität

```
void fieldSensitive() {
   Foo o = new Foo();
   o.field1 = "123";
   o.field2 = source();

   sink(o.field1);
}

class Foo {
   String field1;
   String field2;
}
```

Fig. 7. Feld-Sensitivität

der die App installieren möchte muss dieser die Berechtigung geben auf diese Funktionalitäten zugreifen zu dürfen. Werden die Berechtigungen verweigert, kann die App nicht installiert werden. Das Problem ist, dass oft zu viele Berechtigungen gefordert werden. Entweder aus Unkenntnis oder mit bösen Absichten. Außerdem sind viele Berechtigungen zu mächtig und können ausgenutzt werden. Eine social-network App zum Beispiel, möchte die Berechtigung *android.permission.READ_CONTACTS* um neue Freunde anhand der Email 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. Trojaner können dies ausnutzen.
Um dem entgegenzuwirken kann man die Datenflüsse einer Android App mithilfe der Taint Analyse analysieren. Dabei gibt es einige Besonderheiten und Schwierigkeiten gegenüber anderen *JAVA* Programmen. Eine Android App kann nämlich aus vier verschiedenen Komponenten bestehen.

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

Aufgrund dieser Besonderheit entstehen bestimmte Probleme für eine Taint Analyse. Zum einen hat eine Android App keine zentrale *main* Methode, sondern mehrere Einstiegspunkte. Eine App kann zum Beispiel aus drei Activities und einem Service bestehen. Es gibt zwar eine Haupt-Activity, aber welche Komponente als nächstes an die Reihe kommt ist im allgemeinen nicht bestimmbar, da es oft auf die Benutzereingabe ankommt. 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. Außerdem muss vor der Analyse spezifiziert werden, was Sources und Sinks in Android sind.

4 FlowDroid

FlowDroid ist eine statische Taint Analyse die, Fluss-, Kontext-, Objekt- und Feld-Sensitiv ist [1]. Für die Sinks und Sources benutzt FlowDroid die Ausgabe des Tools SuSi. SuSi ist ein machine-learning Tool, dass vollautomatisch den Android Source Code analysiert und eine Liste von Sinks und Sources generiert. Außerdem wird vor der Analyse ein präziser Android Lebenszyklus modelliert.

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

4.1 Lebenszyklus

Aufgrund der vielen möglichen Einstiegspunkte bei einer Android App, wird bei FlowDroid zunächst eine dummy-main-Mehtode erstellt. Diese Methode wird individuell für jede App erstellt. Die dummy-main enthält nur Teile des Lebenszyklus die auch auftreten können. Um diese zu indentifizieren werden die XML Konfigurationsdateien und der Quellcode analysiert. Danach ruft die dummy-main-Methode nachheinander Teile des Zykluses auf.

4.2 Evaluation

Für die Evaluerung wurde die Testumgebung DroidBench [2] verwenden. DroidBench enthielt zum damaliegen Zeitpunkt 39 Apps mit verschiedenen Analyse Problemen. Vergliechen wurde FlowDroid mit den beiden komerziellen Analyse Tools *App Scan* von IBM und *Fortify SCA* von HP. In dem Evaluierungsbogen (8) ist zu erkennen, dass FlowDroid in den Kategorien *Callbacks* und *Lifecycle* deutlich besser ist, als die beiden anderen Tools. Dies ist auf die präzise Modelierung des Lebenszyklus zurückzuführen. In ein paar Tests schlägt FlowDroid aber auch fehl. Zum Beispiel findet FlowDroid keinen Fehler im *IntentSink1* Test. Das liegt daran, dass in dem Test tainted Daten aus einer Komponente in die nächste übergeben werden und diese die Daten direkt wieder zurück schickt. FlowDroid verfolgt tainted Daten nur innerhalb von Komponenten. Deshalb schlägt der Test hier fehl. Ein weiterer Test der Fehlschlägt ist der *StaticInitialization1* Test. Hier liegt der Grund in dem verwendeten Framework Soot.
Soot ist ein Framework zum analysieren und transformieren von Java und Android Programmen. Es kann aus Java Code einen Call-Graphen generieren. Dabei geht Soot davon aus, dass die komplette statische Initialisierung zu beginn stattfindet, was in diesem Testfall nicht der Fall ist.
Im Ergebniss schneidet FlowDroid jedoch besser ab, als seine beiden Konkurenten. FlowDroid ist mit 86% etwas besser beim Verhältnis zwischen gefunden Fehlern und false-positives (Precision p). Mit 93% gegenüber 50% und 61% ist FlowDroid aber deutlich besser beim auffinden aller Fehler (Recall r).
Mit FlowDroid wurden zusätzlich die damals Top 500 Apps des Google Play Stores getestet. Dabei wurden in den meisten Apps ein Datenleck gefunden.
Außerdem wurden 1000 Malware Apps des VirusShare Projects analysiert. Es wurden pro App ca. zwei Datenlecks gefunden.

5 Ausblick

FlowDroid macht viele richtig, aber wie die Evaluierung gezeigt hat, ist FlowDroid noch nicht perfekt.
Zum Einen sollte die Inter-App-Kommunikation verbessert werden, damit auch tainted Daten zwischen den Komponenten verfolgt werden können. Dazu könnte

[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

Fig. 8. DroidBench Evaluierung, aus [5]

der Sink der einen Komponente gleichzeitig die Source der anderen Komponenten sein. Zum Anderen hat FlowDroid keine Sanitization implementiert. Dies ist jedoch notwendig, damit der Programmierer auf gefundene Fehler reagieren 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. SuSi klassifiziert und kategoriesiert bereits die Sinks und Sources.

References

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/`.
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`.