AppCaulk

Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps

Julian Schütte¹, Dennis Tltze¹, José Maria de Fuentes²

¹ Fraunhofer AISEC, Munich, Germany
² University Carlos III of Madrid, Spain

TrustCom 2014, Beijing
Motivation

- There is no data usage control in Android

- Among 10,000 most popular apps, 5% send out IMEI immediately when started

  ➔ Controlling data flows at application level is required
TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
String imei = tm.getDeviceId();
Uri uri = Uri.parse("http://www.example.com?imei=" + imei);
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);

Static & dynamic data leak detection

- Tracking the **taint state** of registers
- Registers written by a **source** function become tainted with a flag
- **Tainted** registers written to a **sink** function impose a **leak**

Source

```java
TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
String imei = tm.getDeviceId();
Uri uri = Uri.parse("http://www.example.com?imei=" + imei);
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Sink

---

**Static analysis**

- e.g., FlowDroid\(^1\)
- Overapproximative
- Tends to generate false positives

**Dynamic analysis**

- e.g., TaintDroid\(^2\)
- Detects leaks only as they occur
- Requires modified system image

---

\(^1\) [http://sseblog.ec-spride.de/tools/flowdroid/](http://sseblog.ec-spride.de/tools/flowdroid/)

\(^2\) [http://appanalysis.org/](http://appanalysis.org/)
AppCaulk: Overview (1/2)

- Android platform does not provide data flow control
- Static data flow analysis overapproximates
- Simple dynamic taint analysis requires to monitor all registers + modified VM

AppCaulk

- Static data flow analysis to identify call paths of potential leaks
- Injection of a dynamic taint analysis into the app along call paths
- Policy-controlled definition of sources/sinks/countermeasures/…
AppCaulk: Overview (2/2)

Data flow analysis

Instrumentation

Policy

APK

modifiedAPK

Taint analysis & Leak detection

Design Time

Runtime
Efficient Data Flow Analysis (1/2)

- Transformation into \texttt{smali} IR
- Starting at sinks (method name + argument position), mark argument register as \textit{potentially relevant}
- Create slicing, applying propagation logic to registers

- When method parameter is reached, continue with callers
- Stop when no further relevant statements in worklist and taint states did not change since last iteration
Efficient Data Flow Analysis (2/2)

- Backwards slicing creates dfg to all sinks

- Forward slicing (analog to bwd) creates dfg from all sources

- Special cases
  - Writing to static field taints all registers it is assigned to
  - Array indices
Propagation across native methods

- Scope of static analysis: APK bytecode + Android framework.jar
  - Native methods would break taint propagation

- Android 4.3 has ~3600 native methods
- 1339 native methods may propagate data (arguments + return values)
- Many of them are overloaded (e.g., `Math.sqrt(D):D` vs `Math.sqrt(F):F`)
  - Manual definition of native methods propagation rules is feasible.
Propagation across external channels

- Writing tainted data into a file, reading from file → propagate taint flag
- Handled by predefined combinations of channel entry/exit methods

**SQLite DB**

```java
Database.insert(X);
...
String result = Database.query(..);
```

**Intents**

```java
Intent Y = intent.putExtra(String, X);
startActivity(Y);
...
Intent Y = getIntent();
```

**Files**

```java
FileWriter.write(X);
...
FileReader.read(X);
```

**Shared Preferences**

```java
SharedPreferences.editor.put(Y, X);
SharedPreferences.editor.commit();
...
SharedPreferences.getString(Y, X)
```
Instrumentation of Dalvik bytecode

- Add Tracker class
  - Global taint table
  - Handlers for taint propagation
  - Handlers for leak detection
- Represent registers globally unique: Thread id|class|method|register
- For each statement along the call path, insert calls to propagation handler method
Effectiveness evaluation

- Runtime (s) 24
- Median 263
- Avg 7656
- Effectiveness compared against TaintDroid (purely dynamic tainting)
  - Search for leak of `getDeviceID()`: 15 apps relevant and runnable
  - Statically detected leaks not confirmed by TaintDroid: 3/15
  - No misses, no false positives during dynamic test

→ Effectiveness keeps up with purely dynamic taint analysis
Conclusion

- AppCaulk "hardens" Android apps by combining static data flow analysis with injection of a dynamic taint analysis into the app
- Detection rate keeps up with TaintDroid
- Applicable to any Android application
- No modification of Android platform required