FlowDroid

Alex Mariakakis
From CSE 501...again
Motivation

• All sorts of mobile malware exist
  – Selling user information to advertisement/marketing companies
  – Stealing user credentials
  – Premium rate calls and SMS
  – SMS spam
  – Search engine optimization
  – Ransom
Contributions

- **FlowDroid**: the first fully context, field, object and flow-sensitive taint analysis which considers the Android application lifecycle and UI widgets, and which features a novel, particularly precise variant of an on-demand alias analysis.
- **DroidBench**: a novel, open and comprehensive micro benchmark suite for Android flow analyses.
- **Experiments**: demonstrate superior precision and recall to commercial tools and manageable runtimes on real-world apps.
Challenges

1. Multiple entry points

2. Asynchronously executing components

3. Callbacks
Challenges

1. Multiple entry points

2. Asynchronously executing components

3. Callbacks
Challenges

1. Multiple entry points
2. Asynchronously executing components
3. Callbacks
public class LeakageApp extends Activity {
    private User user = null;
    protected void onRestart() {
        EditText usernameText = (EditText) findViewById(R.id.username);
        EditText passwordText = (EditText) findViewById(R.id.pwdString);
        String uname = usernameText . toString();
        String pwd = passwordText . toString();
        if (!uname.isEmpty() && !pwd.isEmpty())
            this.user = new User(uname, pwd);
    }
    // Callback method in xml file
    public void sendMessage(View view) {
        if (user == null) return;
        Password pwd = user.getPwd();
        String pwdString = pwd.getPassword();
        String obfPwd = "";
        // must track primitives
        for (char c: pwdString.toCharArray())
            obfPwd += c + "_"; // String concat
        String message = " User : " + user.getName() + " | Pwd: " + obfPwd;
        SmsManager sms = SmsManager.getDefault();
        sms.sendTextMessage(" +44 020 7321 0905 ", null, message, null, null);
    }
}
Dummy Main Method

LeakageApp la = new LeakageApp();

la.onCreate();

la.onStart();

la.onResume();

p

p

la.sendMessage();

p

la.onPause();

p

la.onStop();

p

la.onRestart();

la.onDestroy();
On-Demand Alias Analysis

```java
tvoid main() {
    Data p = new ...; Data p2 = new ...;
    taintIt(source(), p);
    sink(p.f);
}
tvoid taintIt(String in, Data out) {
    x = out; // x = p → p.f = source()
    x.f = in; // x.f = source()
    sink(out.f); // sink(p.f) → sink(source())
}
```
Context Sensitivity

- Inject context of forward analysis into backward analysis since not all inputs will lead to taints.
  
  Ex: `taintIt(source(), p1) vs. taintIt("public", p2)`

- Whenever an alias is found, work forward from the beginning (rather than backwards) to map taints and avoid unrealizable paths.
Flow Sensitivity

Data p = new ...; Data p2 = p;
sink(p2.f);  
p.f = source();  
sink(p2.f);  

p’s taint not yet activated

not tainted
activation statement
not tainted
tainted

Concept from Andromeda
Evaluation

RQ1: How does FlowDroid compare to commercial taint-analysis tools for Android in terms of precision and recall?

**precision = 86%**

**recall = 93%**
Evaluation

RQ2: Can FlowDroid find all privacy leaks in InsecureBank, an app specifically designed by others to challenge vulnerability detection tools for Android, and what is its performance?

Finds all seven data leaks in 31 seconds
Evaluation

RQ3: Can FlowDroid find leaks in real-world applications and how fast is it?

<table>
<thead>
<tr>
<th>App Source</th>
<th>Run Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Play</td>
<td>Mean &lt; 1 min</td>
<td>Found lots of leaks, claims that most are not malicious</td>
</tr>
<tr>
<td></td>
<td>Max ≈ 4.5 min</td>
<td></td>
</tr>
<tr>
<td>VirusShare Project</td>
<td>Mean = 16 s</td>
<td>Samples were smaller than Google Play apps</td>
</tr>
<tr>
<td></td>
<td>Min = 5 s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Max = 71 s</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation

RQ4: How well does FlowDroid perform when being applied to taint-analysis problems related to Java, not Android, both in terms of precision and recall?

precision = 93%
recall = 97%

<table>
<thead>
<tr>
<th>Test-case group</th>
<th>TP</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aliasing</td>
<td>11/11</td>
<td>0</td>
</tr>
<tr>
<td>Arrays</td>
<td>9/9</td>
<td>6</td>
</tr>
<tr>
<td>Basic</td>
<td>58/60</td>
<td>0</td>
</tr>
<tr>
<td>Collections</td>
<td>14/14</td>
<td>3</td>
</tr>
<tr>
<td>Datastructure</td>
<td>5/5</td>
<td>0</td>
</tr>
<tr>
<td>Factory</td>
<td>3/3</td>
<td>0</td>
</tr>
<tr>
<td>Inter</td>
<td>14/16</td>
<td>0</td>
</tr>
<tr>
<td>Pred</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Reflection</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Sanitizer</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Session</td>
<td>3/3</td>
<td>0</td>
</tr>
<tr>
<td>StrongUpdates</td>
<td>0/0</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>117/121</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 2: SecuriBench Micro test results
Limitations from Implementation

• Rule-based taint propagation for external libraries
  – *E.g.*, adding a tainted element to a set taints the whole set

• Native C calls treated as black box
  – If not predefined rule, assume tainted input leads to tainted output

• Assumes arbitrary, but sequential ordering, so can’t handle multi-threading
Interesting Questions

• Why so much focus on Android? Does it generalize?
• Which do you value more: precision or recall?