Benchmarking Android Security Analysis

A Bachelors Project,
Intermediate Presentation

by Timo Spring
Supervised by Claudio Corrodi
1. Project Motivation

What Is It About?

Problem
- Millions of android apps
- Hundreds of Analyses tools
- Large scale taxonomies classifying them
- Lack of comparison in practice

Project Idea
- Run selected tools on common dataset
- Evaluate the results and compare them
- Draw conclusions why they might be different
2. Tool Selection Process

Literature: Reviewing Types Of Vulnerabilities

Security Smells in Android

Mohammad Ghafari, Pascal Gadjert, Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{ghafari, gadjert, oscar}@inf.unibe.ch

Abstract—The ubiquity of smartphones, and their very broad capabilities and usage, make the security of these devices tremendously important. Unfortunately, despite all progress in security and privacy mechanisms, vulnerabilities continue to proliferate. Research has shown that many vulnerabilities are due to insecure programming practices. However, each study has often dealt with a specific issue, making the results less actionable for practitioners.

To promote secure programming practices, we have reviewed related research, and identified avoidable vulnerabilities in Android-run devices and the security code smells that indicate their presence. In particular, we explain the vulnerabilities, their corresponding smells, and we discuss how they could be eliminated or mitigated during development. Moreover, we develop a lightweight static analysis tool and discuss the extent to which it successfully detects several vulnerabilities in about 46,000 apps hosted by the official Android market.

I. INTRODUCTION

Given these premises, the primary goal of this work is to shed light on the root causes of programming choices that compromise users’ security. In contrast to previous research that has often dealt with a specific issue, we study this phenomenon from a broad perspective. We introduce the notion of security code smells i.e., symptoms in the code that signal the prospect of a security vulnerability. We have identified avoidable vulnerabilities, their corresponding smells in the code; and discuss how they could be eliminated or mitigated during development. We have also developed a lightweight static analysis tool to look for several of the identified security smells in 46,000 apps. In particular, we answer the following three research questions:

- RQ1: What are the security code smells in Android apps?
  We have reviewed major related work, especially those...
2. Tool Selection Process

Literature: Reviewing Benchmarking Process

SCAM 2017

Security Smells in

Mohammad Ghafari, Pascal Gadient, Oscar Ødegaard
Software Composition Group, University of Bern, Switzerland
{ghafari, gadient, oscar}@inf.unibe.ch

Abstract—The ubiquity of smartphones, and their very broad capabilities and usage, make the security of these devices tremendously important. Unfortunately, despite all progress in security and privacy mechanisms, vulnerabilities continue to proliferate.

Research has shown that many vulnerabilities are due to insecure programming practices. However, each study has often dealt with a specific issue, making the results less actionable for practitioners.

To promote secure programming practices, we have reviewed related research, and identified avoidable vulnerabilities in Android-run devices and the security code smells that indicate their presence. In particular, we explain the vulnerabilities, their corresponding smells, and we discuss how they could be eliminated or mitigated during development. Moreover, we develop a lightweight static analysis tool and discuss the extent to which it successfully detects several vulnerabilities in about 46,000 apps hosted by the official Android market.

I. INTRODUCTION

Given the widespread adoption of smartphones, the need to ensure their security has become increasingly important. However, despite the widespread use of security mechanisms, vulnerabilities continue to emerge.

This study identifies avoidable vulnerabilities in Android applications and the corresponding security code smells. We explain the vulnerabilities, their corresponding smells, and discuss strategies for eliminating or mitigating them during development.

Moreover, we developed a lightweight static analysis tool and evaluated its effectiveness in detecting several vulnerabilities in about 46,000 apps hosted on the official Android market.

MUBench: A Benchmark for API-Misuse Detectors

Sven Amann1, Sarah Nadi2, Hoan A. Nguyen3, Tien N. Nguyen3, Mira Mezini1

1Technische Universität Darmstadt, 2Lancaster University, 3Iowa State University
{amann, nadi, mezini}@cs.tu-darmstadt.de, {hoan, tien}@iastate.edu

ABSTRACT

Over the last few years, researchers have developed a multitude of automated bug-detection approaches that mine a class of bugs that we call API misuse. Evaluations on a variety of software products show the omnipresence of such misuse and the ability of the approaches to detect them.

This work presents MUBench, a dataset of 89 API misuse that we collected from 33 real-world projects and a survey. We develop a lightweight static analysis tool and empirically analyze the prevalence of API misuses compared to other types of bugs, finding that they are rare, but almost always cause crashes. Furthermore, we discuss how to use it to benchmark and compare API-misuse detectors.

CCS Concepts

•Software and its engineering → Software defect analysis; Software post-development issues;

2016 IEEE/ACM 13th Working Conference on Mining Software Repositories
2. Tool Selection Process

Literature: Reviewing Ground Concepts

A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks

Siegfried Rasthofer & Steven Arzt
Secure Software Engineering Group
EC SPRIDE, Technische Universität Darmstadt
{firstname.lastname}@ec-spride.de

Eric Bodden
Secure Software Engineering Group
Fraunhofer SIT & Technische Universität Darmstadt
eric.bodden@sit.fraunhofer.de

Abstract—Today’s smartphone users face a security dilemma: many apps install operate on privacy-sensitive data, although they might originate from developers whose trustworthiness is hard to judge. Researchers have addressed the problem with more and more sophisticated static and dynamic monitoring tools as an aid to assess how apps use private user data. Those tools, however, rely on the manual configuration of lists of sources of sensitive data as well as sinks which might leak data to untrusted observers. Such lists are hard to come by.

We thus propose S:Si, a novel machine-learning guided approach for identifying sources and sinks directly from the code of any Android API. Given a training set of hand-annotated sources and sinks, S:Si identifies other sources and sinks in the entire API. To provide more fine-grained information, S:Si further categorizes the sources (e.g., unique identifier, location information, etc.) and sinks (e.g., network, file, etc.).

For Android 4.2, S:Si identifies hundreds of sources and sinks with over 92% accuracy, many of which are mined by current information-flow tracking tools. An evaluation of about 11,000 malware samples confirms that many of these sources and sinks are indeed used. We furthermore show that S:Si can reliably classify sources and sinks even in new, previously unseen Android versions and components like Google Glass or the Chrome APL.

This work focuses on Android. As we show, existing analysis tools, both static and dynamic, focus on a handful of hand-picked sources and sinks, and can thus be circumvented by malicious applications with ease. It would be too simple, though, to blame the developers of those tools. Android’s version 4.2, for instance, comprises about 110,000 public methods, which makes a manual classification of sources and sinks clearly infeasible. Furthermore, each new Android version includes new functionality (e.g., NFC in Android 2.3 or Restricted Profiles...
2. Tool Selection Process

Literature: Reviewing Different Tools
2. Tool Selection Process

Literature: Reviewing Robustness Of Tools

RUHR-UNIVERSITÄT BOCHUM
Horst Görtz Institute for IT Security

Technical Report TR-HGI-2016-003

Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation
Johannes Hoffmann, Teemu Rytilähti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, Thorsten Holz

Chair for Systems Security

2016 IEEE European Symposium on Security and Privacy

HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving
G. Calzavara (Italy), Ilya Grishchenko, Matteo Maffei
CISPA, Saarland University

2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Composite Constant Propagation: Application to Avoid Inter-Component Communication Analysis
Octeau, Luchaup, Dering, Jha, and McDaniel
Department of Computer Sciences, University of Wisconsin
Department of Computer Science and Engineering, Pennsylvania State University

Many program analyses require statically inferring values of composite types. However, current analyzers do not account for correlations between object fields. In this paper, we introduce new composite constant propagation, which we develop the solver that infers all possible values of composite types, including procedural, flow and context-sensitive sensitivities. In this paper, we introduce a new composite constant propagation, which is parameterized from abstract syntax trees.

Unfortunately, existing studies of application interfaces are such as information flow analysis [22], [24], [28], [29], patch generation for privilege escalation vulnerabilities [42] and detection of stealthy behavior [18].
2. Tool Selection Process

Literature: Major Contribution
2. Tool Selection Process

Focus On Vulnerability Detection

2. Tool Selection Process

   - Not Found
2. Tool Selection Process

- No Tools
2. Tool Selection Process

- Not Reachable Researcher
2. Tool Selection Process

- Access Refused
2. Tool Selection Process

- Unresponsive Researcher

Amandroid, ApkCombiner, AppGuard, Bagheri, ConDroid, CRePE, DexDiff, Flowdroid, Geneiatakis, Grab’nRun, NoFrak, Stowaway,
2. Tool Selection Process

*Responsive Researcher*

Amandroid, ApkCombiner, AppGuard, AppCaulk, AppCracker, AppFence, AppAudit, AppProfiler, AppSealer, AppRay, Aquifer, ASM, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, CredMiner, CryptoLint, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, Flowdroid, Gallo, Geneiatakis, Grab’nRun, HornDroid, IccTA, ICTAL, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermutationFlow, Poeplau, Quire, Ren, SCAnDroid, Scoria, SEFA, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo
2. Tool Selection Process

Focus On Information Disclosure

Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bifocals, Buhov, Buzzer, CMA, CredMiner, CryptoLint, Desnos, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, Flowdroid, Geneiatakis, Grab’nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, SMV-HUNTER, STAMBA, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo
2. Tool Selection Process

And The Winners Are…
2. Tool Selection Process

… But, Remove Tools That Cannot Be Setup

ApkCombiner, AppGuard, AppCaulk, Bagheri, COVERT, Enck, Epicc, HornDroid, IccTA, Smith,
2. Tool Selection Process

... And Those That Cannot Be Analysed
### 3. Selected Tools In A Nutshell

**In A Nutshell – Pretty Much The Same**

<table>
<thead>
<tr>
<th></th>
<th>COVERT</th>
<th>Flowdroid</th>
<th>IccTA</th>
<th>IC3 (Epicc)</th>
<th>Horndroid</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong></td>
<td>Static &amp; Formal</td>
<td>Static</td>
<td>Static</td>
<td>Static</td>
<td>Static &amp; Formal</td>
</tr>
<tr>
<td><strong>Artefact:</strong></td>
<td>Manifest</td>
<td>Manifest Code (native)</td>
<td>Manifest Layout</td>
<td>Manifest</td>
<td>Code (reflective)</td>
</tr>
<tr>
<td><strong>Data Structure:</strong></td>
<td>Call Graph</td>
<td>Call Graph</td>
<td>Call Graph</td>
<td>Call Graph</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>CFG</td>
<td>CFG</td>
<td>CFG</td>
<td>CFG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ICFG</td>
<td>ICFG</td>
<td>ICFG</td>
<td>ICFG</td>
<td></td>
</tr>
<tr>
<td><strong>Code Representation</strong></td>
<td>Jimple</td>
<td>Jimple</td>
<td>Jimple</td>
<td>Jimple</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Sensitivity:</strong></td>
<td>Flow Context</td>
<td>Flow Context</td>
<td>Flow Context</td>
<td>Flow Context</td>
<td>N/A</td>
</tr>
</tbody>
</table>

3. Selected Tools In A Nutshell

Results Hard To Find, To Read And To Understand…

XML file content:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<analysisReport>
  <name>apksToTest</name>
  <apps/>
  <vulnerabilities>
    <vulnerability>
      <type>Intent Spoofing</type>
      <description>App org.cert.sendsms puts data (retrieved from an Explicit Intent (Component = MainActivity)) on an unsafe sink (SMS MMS) in one of its components (org.cert.sendsms.MainActivity). A malicious app can send a sensitive data from this channel.</description>
    </vulnerability>
    <vulnerabilityElements>
      <type>APP</type>
      <description>org.cert.sendsms</description>
      <element>
        <type>COMPONENT</type>
        <description>org.cert.sendsms.MainActivity</description>
      </element>
      <element>
        <type>INTENT</type>
        <description>Explicit Intent (Component = MainActivity)</description>
        <elementLabel>i2</elementLabel>
      </element>
      <element>
        <type>METHOD</type>
        <description>org.cert.sendsms.MainActivity: void onActivityResult(int, int, android.content.Intent)</description>
      </element>
      <element>
        <type>SINK_TYPE</type>
        <description>SMS_MMS</description>
      </element>
    </vulnerabilityElements>
  </vulnerabilities>
</analysisReport>
```
3. Selected Tools In A Nutshell

Results Hard To Find, To Read And To Understand...

Running data flow analysis...
Found dex file 'classes.dex' with 456 classes in '/Users/timespring/Desktop/droid-Security-Thesis/apk_sample/validation apk/SENDSMS.apk'
[Call Graph] For information on where the call graph may be incomplete, use the verbose option to the cg phase.
[Spark] Pointer Assignment Graph in 0.0 seconds.
[Spark] Type masks in 0.0 seconds.
[Spark] Pointer Graph simplified in 0.0 seconds.
[Spark] Propagation in 0.1 seconds.
[Spark] Solution found in 0.1 seconds.
Callback analysis done.
Found 1 layout controls in file res/layout/activity_main.xml
[Call Graph] For information on where the call graph may be incomplete, use the verbose option to the cg phase.
[Spark] Pointer Assignment Graph in 0.0 seconds.
[Spark] Type masks in 0.0 seconds.
[Spark] Pointer Graph simplified in 0.0 seconds.
[Spark] Propagation in 0.0 seconds.
[Spark] Solution found in 0.0 seconds.
Running incremental callback analysis for 1 components...
Incremental callback analysis done.

Found a flow to sink \texttt{virtual invoke} $r3.<\texttt{pro.cert.sendsms.MainActivity}: \texttt{void startActivityForResult} (\texttt{android.content.Intent}: \texttt{int})\rightarrow$($r2$, $0$), from the following sources:
- $r2 = \texttt{virtual invoke} \texttt{r5}.<\texttt{android.telephony.TelephonyManager}: \texttt{java.lang.String} \texttt{getDeviceId}()$() \in <\texttt{pro.cert.sendsms.ButtonListener}: \texttt{void onClick} (\texttt{android.view.View})>

Found a flow to sink \texttt{virtual invoke} $r3.<\texttt{android.telephony.SmsManager}: \texttt{void sendTextMessage} (\texttt{java.lang.String}, \texttt{java.lang.String}, \texttt{java.lang.String}, \texttt{android.app.PendingIntent}, \texttt{android.app.PendingIntent})\rightarrow$($1224567890$, null, $r3$, null, null), from the following sources:
- $r3 = \texttt{parameter} 2: \texttt{android.content.Intent}$ \in <\texttt{pro.cert.sendsms.MainActivity}: \texttt{void startActivityForResult} (\texttt{int}, \texttt{int}, \texttt{android.content.Intent})>

Found a flow to sink \texttt{static invoke} $\texttt{android.util.Log}: \texttt{int} \in (\texttt{java.lang.String}, \texttt{java.lang.String})\rightarrow$($\texttt{SendSMS}$, $r6$), from the following sources:
- $r6 = \texttt{virtual invoke} \texttt{r5}.<\texttt{android.telephony.TelephonyManager}: \texttt{java.lang.String} \texttt{getDeviceId}()$() \in <\texttt{pro.cert.sendsms.ButtonListener}: \texttt{void onClick} (\texttt{android.view.View})>

Analysis has run for 6.296900003 seconds
3. Selected Tools In A Nutshell

Results Hard To Find, To Read And To Understand…
3. Selected Tools In A Nutshell

*It’s All About Sources And Sinks*

- Sources: `getDeviceId()`
- Sensitive Data: e.g. UID
- Sinks: `sendTextMessage(uid)`
- Leak

Is it only a question of who has the best sources and sinks list?
3. Selected Tools In A Nutshell

Own Implementation Runs Tools And Parses Output

Component: android.util.Log
Class: org.cert.sendsms.Button1Listener
Method: void onClick(android.view.View)
Line: 25
Detected by: flowdroid, iccta

Problem: Only class and method are reported by all tools
4. Case Study: SendSMS App

Run tools on app with known vulnerabilities

- SendsSMS.apk with known inter-app communication vulnerabilities
- Gets the UID and sends it over SMS and writes it to log file
4. Case Study: SendSMS App

AppLeaks The UID Over SMS And To The Log File

Component: Button1Listener

```java
onClick(View arg0)
23 String uid = tManager.getDeviceId(); // SOURCE
25 Log.i("SendSMS: ", "DeviceId "+uid); // SINK
26 this.act.startActivityForResult(i, 0); // SINK
```

Component: MainActivity

```java
sendSMSMessage(String message)
52 smsManager.sendTextMessage("1234567890", message); //SINK
```
4. Case Study: SendSMS App

*Especially HornDroid Seems To “Over-Report”*

<table>
<thead>
<tr>
<th></th>
<th>True Positives</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Log.i(uid)</td>
<td></td>
</tr>
<tr>
<td>Flowdroid</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>IccTA</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>HornDroid</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>COVERT</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>IC3</td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>startActivity(Intent)</th>
<th>sendSMS(uid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowdroid</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>IccTA</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>HornDroid</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>COVERT</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>IC3</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Log.v(String)</th>
<th>Log.i(String)</th>
<th>Log.i(String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowdroid</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IccTA</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>HornDroid</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>COVERT</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>IC3</td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
4. Case Study: SendSMS App

**Flowdroid Might Be Biased**

<table>
<thead>
<tr>
<th></th>
<th>True Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Log.i(uid)</td>
</tr>
<tr>
<td>Flowdroid</td>
<td>⬜</td>
</tr>
<tr>
<td>IccTA</td>
<td>⬜</td>
</tr>
<tr>
<td>HornDroid</td>
<td>⬜</td>
</tr>
<tr>
<td>COVERT</td>
<td>⬜</td>
</tr>
<tr>
<td>IC3</td>
<td>⬜</td>
</tr>
</tbody>
</table>

Attention:
Might be biased! Same creators as DroidBench. Claim to have 86% precision.
4. Case Study: SendSMS App

Vulnerability Detected, But Not As Precise As Others

Report:

```java
onActivityResult(int, int, Intent)
37 sendSMSMessage(data.getExtras().getString("secret"));
```

Instead of:

```java
sendSMSMessage(String message)
52 smsManager.sendTextMessage("1234567890", null, message, null, null); //SINK
```
4. Case Study: SendSMS App

HornDroid Is The Only Tool Reporting False Positives

Report:

```java
onActivityResult(int, int, Intent)
36   Log.v("In SendSMS: ", "Data received");
...
40   Log.i("In SendSMS: ", "Data received");
...
44   Log.i("In SendSMS: ", "No data received");
```

False Positives

- Log.v(String)
- Log.i(String)
- Log.i(String)
4. Case Study: SendSMS App

There Are Differences In What The Tools Report

- FlowDroid and IccTA found all leaks without false alarm
- HornDroid found most leaks, but reports many false positives
- COVERT and IC3 only found part of the leaks

Problem: Analysis for false positives not scalable!
5. Benchmarking Concept

How To Include Both Worlds

Small scale qualitative
- DroidBench dataset (~30 apps)
- Manually check for false positives and false negatives

Large scale quantitative
- F-Droid dataset (~2.6k apps)
- Automatically analyse number of detections and matchings
6. Lessons Learned So Far

*It’s Tricky!*

- Hundreds of tools, but only few are actually available and can be setup
- Tools are not user-friendly and results poorly documented
- All of them claim to be the best – we’ll see about that …
7. Outlook

What’s Next?

- Fine tune automatic analysis (refactoring)
- Check DroidBench dataset
- Run automatic analysis on F-droid dataset
- Evaluation of results (quantitative)
- Draw conclusions