Code static analysis for security

ISEL – Instituto Superior de Engenharia de Lisboa
Rua Conselheiro Emídio Navarro, 1 | 1959-007 Lisboa

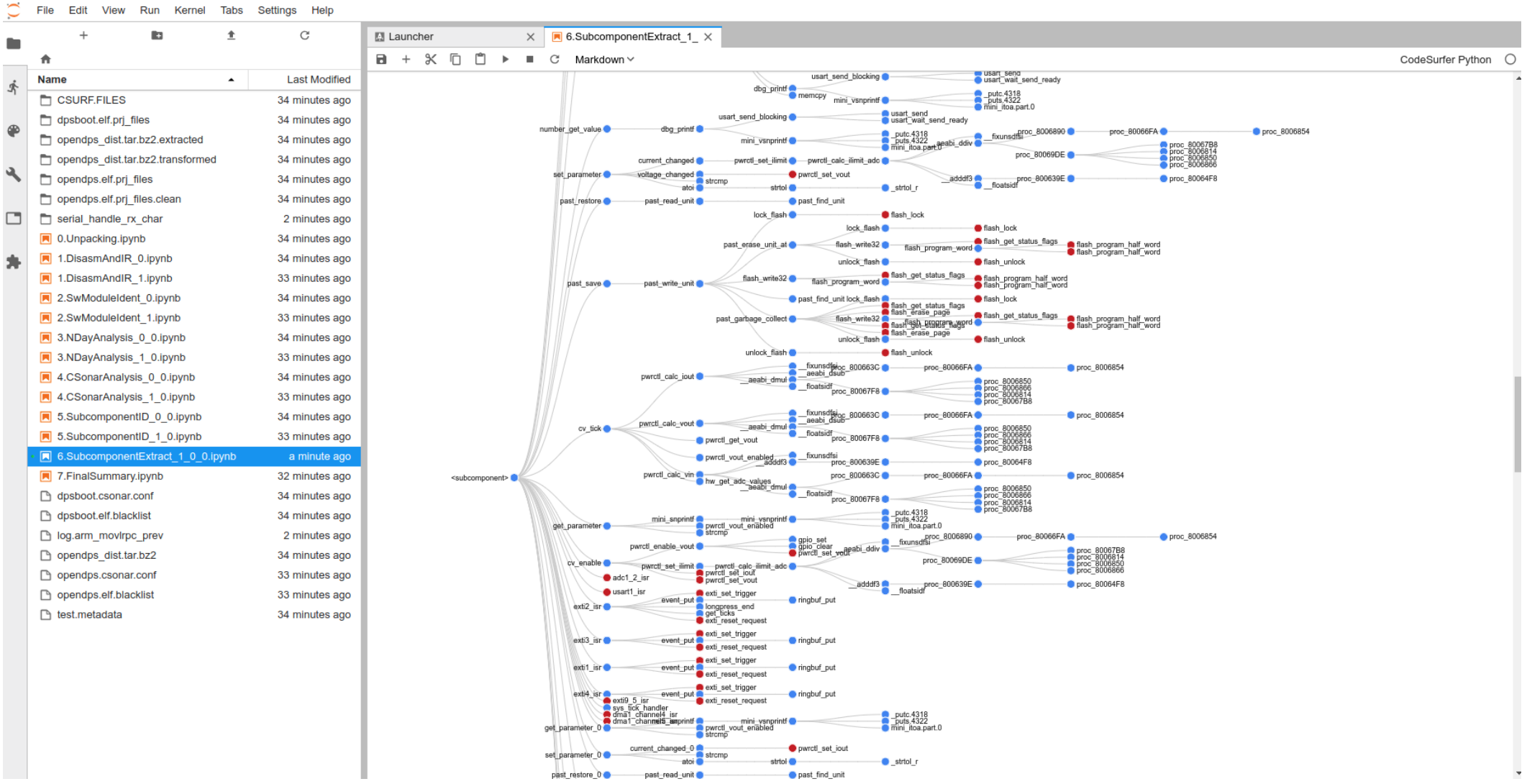
Agenda

- How to find bugs / vulnerabilities in code?
- Manual code analysis
- Automatic code analysis
  - symbolic analysis
  - semantic analysis
    - control flow
    - Data flow
- Examples of Static Application Security Testing Tools (SAST)

Manual code analysis

- Code Understanding Strategies
  - follow malicious entries
  - Analyze by module or algorithm
- Candidate Point Strategies
  - Generic candidate search strategy
  - Identify simple candidate points with tools like grep or findstr
  - Track the result of attack injection

A tool to navigate the code



How to do automatic code analysis?

- Objective: Identify vulnerabilities in code by looking for vulnerable functions
- Unitary tests?
  - 90% of errors involve interactions of multiple functions
  - Maybe that's why the programmer didn't detect the problem
  - Errors occur under various conditions that are difficult to reproduce in tests.
- Naive approach
  - grep gets *.c
  - It forces the testers to know all the vulnerable functions
  - You must manually test for each function.
  - Does not distinguish calls to functions to function names written in strings or comments

```c
#include <stdio.h>
int main(int argc, char **argv)
{
    char buf[8]; // buffer for eight characters
    gets(buf); // read with gets (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0; // 0 as return value
}
```

Tools for Static Aplication Security Testing (SAST)

- The purpose of SAST tools is to analyze source code and provide reports on vulnerabilities found, helping to decide on:
  - Which elements were detected that are not vulnerabilities
  - Which elements are at acceptable risk and therefore are not immediately addressed
  - What are the elements to be mitigated and how to do it
- False negatives
  - They don't detect all situations
  - Limited by the database
  - It is not possible to test all conditions in good time.
- False positives
  - They signal situations that are not vulnerabilities

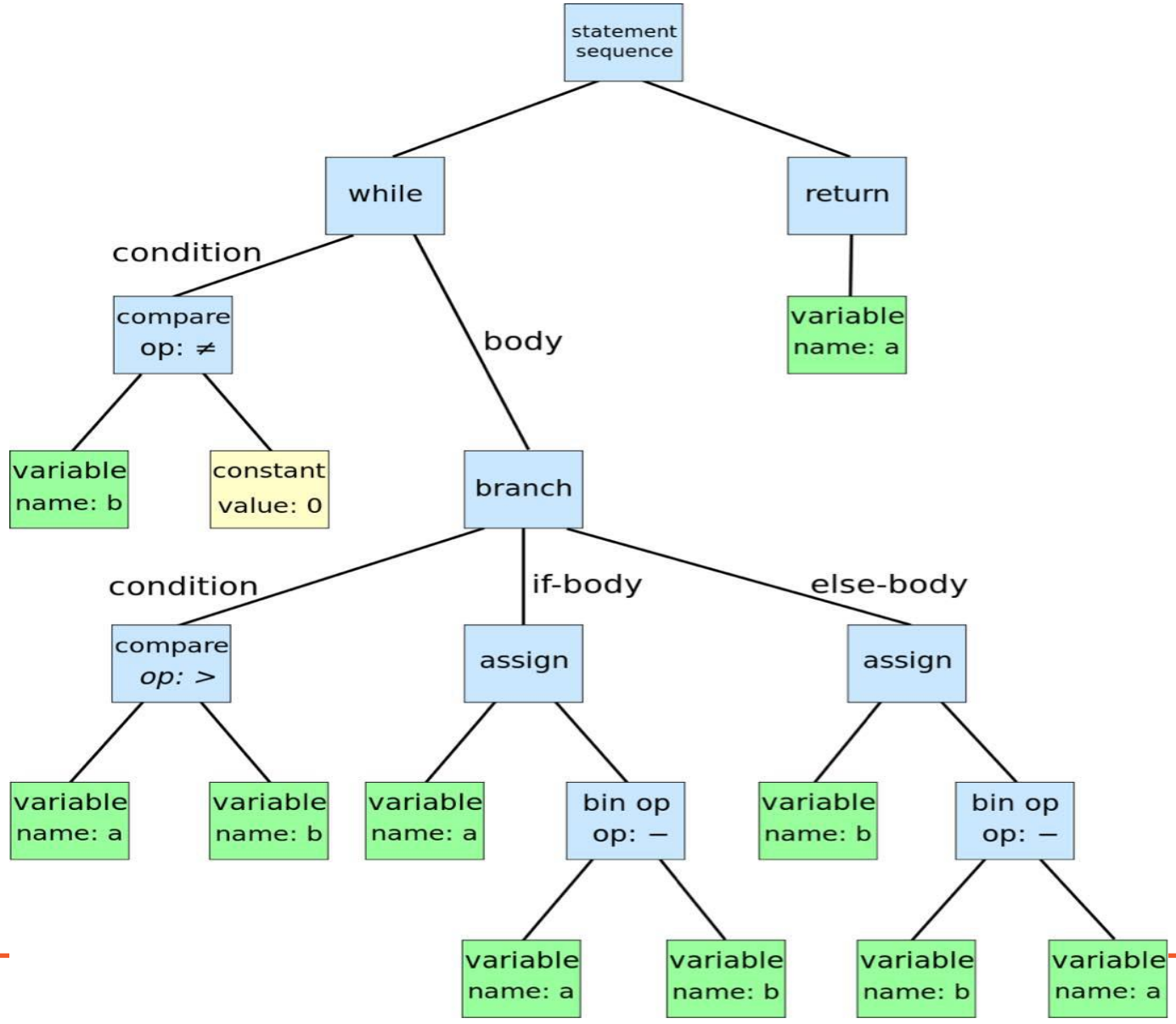
Tools for Static Aplication Security Testing (SAST)

- Lexic analyzers (symbolic)
- Operate on the words generated by the scanner
- Don't confuse ***getshow*** with ***gets*** function
- Semantic analyzers
  - Operate on the abstract syntax tree generated by the parser
  - Don't confuse ***gets*** variable with ***gets*** function call
- The semantic analysis is organized in
  - Type checking
  - Flow control analysis
  - Data flow analysis

Semantic analysis

- SAST tools have similarities to a compilation of programming languages
  - Phases in a compiler:
    Source code -> lexic analysis -> pre-processing -> Sintatic and Semantic analysis -> Intermediate Code Generation -> Optimization -> Machine Code Generation -> Machine Code
- Lexic analysis separates text into relevant tokens
- The third phase builds the syntax tree based on the grammar of the language

Abstract Sintax Tree



https://en.wikipedia.org/wiki/Abstract_syntax_tree

Example of a tool for static analysis

- Example
  - RATS: https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml
  - C, C++, Perl, PHP, Python
- Search for vulnerable functions
- Use database with vulnerabilities

Function	Potential vulnerability	Solution	Risk
`access`	May lead to improper file access under race conditions	Manipulate descriptors and not symbolic names if possible	Average
`fread`	May lead to input with malicious effects	Check input	Low
`fscanf`	May result in buffer overflow	Use precision specifiers	High

Semantic analysis

Examples of Semantic Analysis with Type Checking

- Signal error
  - Signed integer is assigned to unsigned value or vice versa
- Truncation error
  - Integer represented with a given number of bits is assigned to a variable with fewer bits
- Some tools make it possible to annotate code with contracts that can be verified
  - SAL - Standard Annotation Language (Microsoft), C and C++

```
_checkReturn void *__cdecl malloc(__in size_t _Size);
```

  - JIF - Java + Information Flow (Cornell)

```
int {Alice → Bob} x;
```

Semantic analysis

Flux control

- Test multiple execution paths for errors

```
1   char * reserve_memory(int size) {
2       char *memory;
3       if (size > 0)
4           memory = (char*) malloc(size);
5       if (size == 1)
6           return NULL;
7       memory[0] = 0;
8       return memory;
9   }
```

- Several possible paths <2,3,5,7,8>, <2,3,4,5,7,8> and <2,3,4,5,6>
- Build flow control graph based on different conditions, cycles and function calls

Semantic analysis

Flux control

- Flow control analysis involves a representative set of paths.
  - For each path, the route is simulated, generating alarms when relevant
  - In general it is not possible to test with concrete values but with relevant cases

    size=1, size>0, size<>1, size <=0
- In the example there are errors

  2,3,5,7,8 (array not started)

  2,3,4,5,6 (memory not freed)

```
1   char * reserve_memory(int size) {
2       char *memory;
3       if (size > 0)
4           memory = (char*) malloc(size);
5       if (size == 1)
6           return NULL;
7       memory[0] = 0;
8       return memory;
9   }
```

Semantic Analysis

Data flow

- Compromise Analysis – Checks whether sensitive functions are at risk of using compromised or private data
- Tools that use this technique mark data as:
  - **`tainted`** – indication of compromised potential value; for example values taken from inputs, scanf or gets functions in C
  - **`untainted`** – indication that the value (eg parameter of a function) cannot be compromised; for example formatting strings from the printf function in C.
- Examples of Rules for Propagating Compromise
  - Variable a is compromised, **`b=a`** results in **`b`** compromised
  - **`d = f(a)`** may or may not lead to compromise of **`d`** (function **`f`** may internally call another one that cleans the input)

NIST 500-268

- https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analysis
  - Published by the North American agency “National Institute of Standards and Technology”
- The tools must:
  - Identify all classes of vulnerabilities listed in Annex A
  - Report verbatim any vulnerabilities you have identified
  - For any vulnerabilities not listed in Appendix A, report the class using a semantically equivalent name
  - For any identified vulnerabilities, report at least one location providing the directory path, filename and line number
  - Identify vulnerabilities despite the coding complexity listed in Appendix B
  - Have an acceptably low false positive rate.

NIST 500-268 – annex A

Name	CWE ID	Description	Language(s)	Relevant Complexities
Input Validation				
Basic XSS	80	Inadequately filtered input, allows a malicious script to be passed to a web application that in turn passes it to another client.	C,C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
Resource Injection	99	Inadequately filtered input is used in an argument to a resource operation function.	C, C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
OS Command Injection	78	Inadequately filtered input is used in an argument to a system operation execution function.	C, C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type
SQL Injection	89	Inadequately filtered input is used in an argument to a SQL command calling function.	C, C++, Java, SPARK	taint, scope, address alias level, container, local control flow, loop structure, buffer address type

NIST 500-268 – annex B

Complexity	Description	Enumeration

...

Scope	scope of control flow related to weakness	local, within-file/inter-procedural, within-file/global, inter-file/inter-procedural, inter-file/global, inter-class
Taint	type of tainting to input data	argc/argv, environment variables, file or stdin, socket, process environment

SAST Tool for Web application

- WAP – static code analysis tool aimed at web applications written in PHP
  - http://awap.sourceforge.net/
- Three Entities: Inputs, Sensitive Functions, and Sanitation Functions
  - Entries: `$_GET`, `$_POST`
  - Sensitive functions: `mysql_query`
  - Sanitation functions: `mysql_real_escape_string` (makes string sanitized so they can be passed to `mysql_query function`)
- Compromise propagation rules
  - If the data stream starts from an input and arrives at a sensitive function **without going** through a sanitization function **=> vulnerability**
  - If the data flow starts from an input and arrives at a sensitive function **passing through an adequate sanitization function => there is no vulnerability**

Use case: Android

```
1  public class LeakageApp extends Activity{
2  private User user = null;
3  protected void onRestart(){
4    EditText usernameText =
       (EditText)findViewById(R.id.username);
5    EditText passwordText =
       (EditText)findViewById(R.id.pwdString);
6    String uname = usernameText.toString();
7    String pwd = passwordText.toString();
8    if(!uname.isEmpty() && !pwd.isEmpty())
9      this.user = new User(uname, pwd);
10 }
11 //Callback method in xml file
12 public void sendMessage(View view){
13   if(user == null) return;
14   Password pwd = user.getpwd();
15   String pwdString = pwd.getPassword();
16   String obfPwd = "";
17   //must track primitives:
18   for(char c : pwdString.toCharArray())
19     obfPwd += c + "_"; //String concat.
20
21   String message = "User: " +
22      user.getName() + " | Pwd: " + obfPwd;
23   SmsManager sms = SmsManager.getDefault()
24   sms.sendTextMessage("+44 020 7321 0905",
25     null, message, null, null);
26 }
```

- Android code example leaking information
- Line 5: reads password from text box marked as to enter passwords
- Line 24: send password by SMS

SAST for Android: FlowDroid

- Static Analysis Tool for Android Applications

- It takes into account all application context, objects, fields and compromise analysis that considers the lifecycle of Android applications

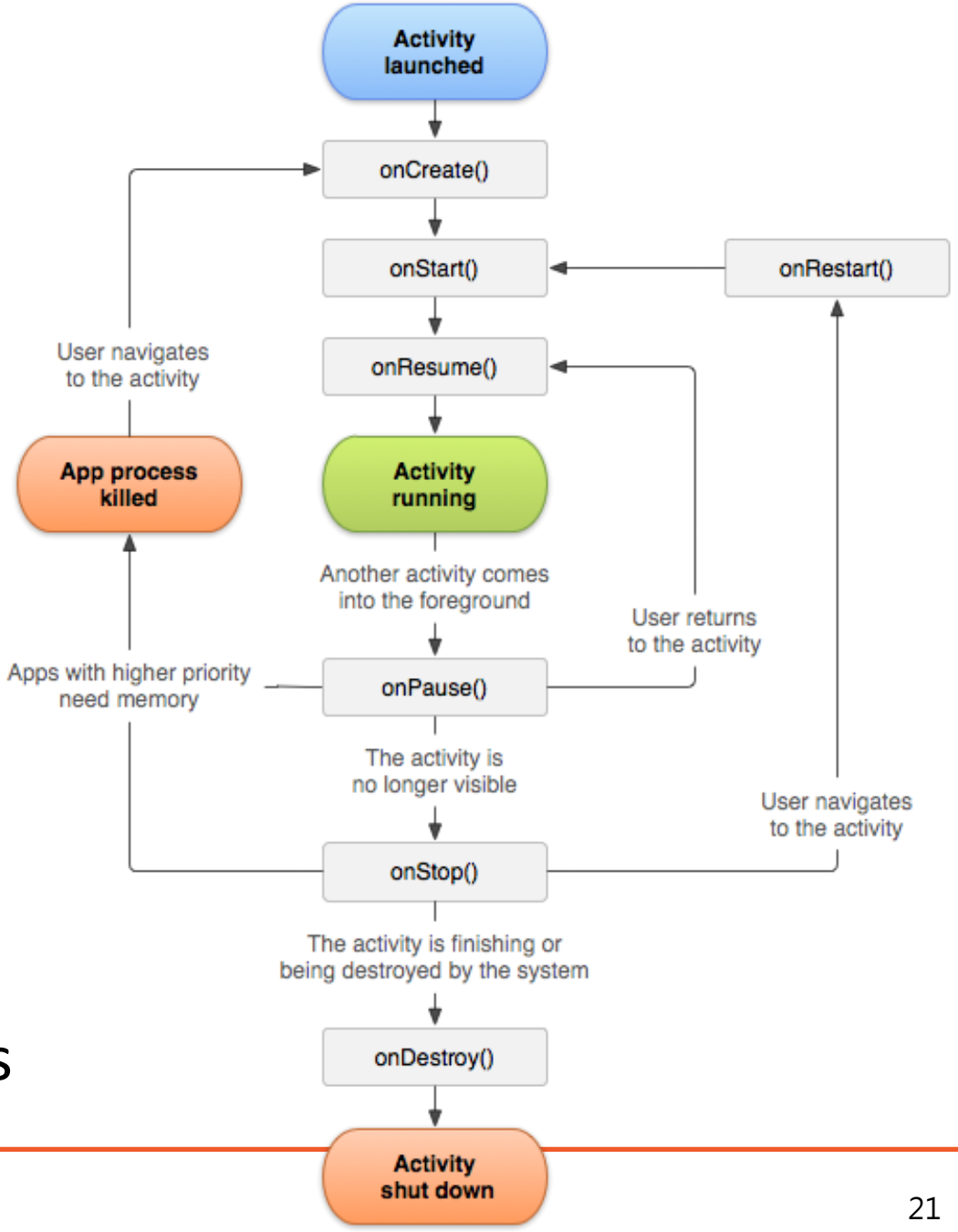
- It scans the application's bytecode and configuration files to find potential data leaks

- Paper: https://www.bodden.de/pubs/far+14flowdroid.pdf

  “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps”, PLDI 2014

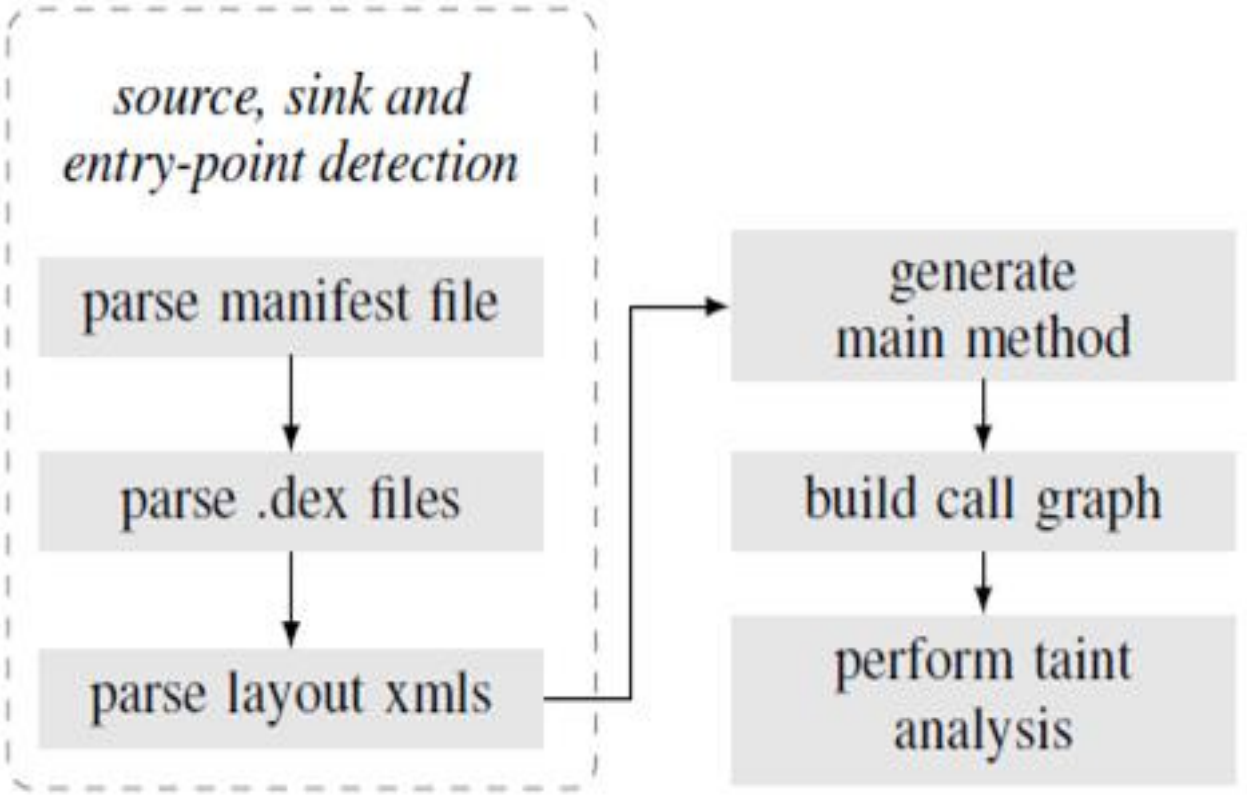
Challenges in FlowDroid

- Need a model for the lifecycle of Android applications
  - Lifecycle driven by callbacks, there is no single entry point
- It is not possible to determine sensitive information by code alone.
  - Example of text box that receives password (android:password)
- Java applications have a non-trivial execution model, with dynamic method dispatch and aliasing
- High rate of false positives and false negatives



Funcionamento geral

- Looks for actors in the application lifecycle, as well as calls to critical **sources** and **sinks**
- Generates a main method from the methods identified in the lifecycle, which is used for a flow control analysis.
- At the end, the paths that lead information from sources to critical destinations are identified



Example

- Two source methods
  - Line 9: Call to `getCid()`, which returns the GSM cell ID.
  - Line 11: call to `getLac()`, which returns the GSM location code
- Together, these two pieces of information can be used to identify the GSM cell tower
- Line 12: the code tests whether the device is at a certain location in Berlin

```
1  void onCreate() {
2  TelephonyManager tm; GsmCellLocation loc;
3  // Get the location
4  tm = (TelephonyManager) getContext().
5      getSystemService
           (Context.TELEPHONY_SERVICE);
6  loc = (GsmCellLocation)
       tm.getCellLocation();
7
8  //source: cell-ID
9  int cellID = loc.getCid();
10 //source: location area code
11 int lac = location.getLac();
12 boolean berlin = (lac == 20228 && cellID
       == 62253);
13
14 String taint = "Berlin: " + berlin + " ("
       + cellID + " | " + lac + ")";
15 String f = this.getFilesDir() +
       "/mytaintedFile.txt";
16 //sink
17 FileUtils.stringToFile(f, taint);
18 //make file readable to everyone
19 Runtime.getRuntime().exec("chmod 666 "+f);
20 }
```

Listing 1: Android Location Leak Example

Examples of *sources* and *sinks*

- <android.bluetooth.BluetoothAdapter: java.lang.String getAddress()> -> _SOURCE_
- <android.net.wifi.WifiInfo: java.lang.String getMacAddress()> -> _SOURCE_
- <android.telephony.gsm.GsmCellLocation: int getCid()> -> _SOURCE_
- <android.telephony.gsm.GsmCellLocation: int getLac()> -> _SOURCE_
- <org.apache.http.impl.client.DefaultHttpClient: org.apache.http.HttpResponse execute(org.apache.http.client.methods.HttpUriRequest)> -> _SINK_
- <android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app.PendingIntent)> android.permission.SEND_SMS -> _SINK_
- <android.telephony.SmsManager: void sendMultipartTextMessage(java.lang.String,java.lang.String,java.util.ArrayList,java.util.ArrayList,java.util.ArrayList)> android.permission.SEND_SMS -> _SINK_

FlowDroid and DroidBench usage

- https://github.com/secure-software-engineering/FlowDroid

- A suite of test applications, containing 39 purpose-written applications with sensitive data leaks

  https://github.com/secure-software-engineering/DroidBench

- Example

```
java -jar soot-infoflow-cmd-jar-with-dependencies.jar -p
C:\Users\josem\AppData\Local\Android\Sdk\platforms\android-29\android.jar -s
FlowDroid\soot-infoflow-android\SourcesAndSinks.txt -a
DroidBench\apk\InterAppCommunication\SendSMS.apk
```

Results

- The ***sink*** staticinvoke Log.i(java.lang.String, java.lang.String)>("SendSMS: ", …) in method <org.cert.sendsms.Button1Listener: void onClick(android.view.View)> was called with values from the following ***sources***:
  - virtualinvoke $r5.<**android.telephony.TelephonyManager: java.lang.String getDeviceId()**>() in method <org.cert.sendsms.Button1Listener: void onClick(android.view.View)>
- The ***sink*** MainActivity.startActivityForResult(android.content.Intent,int) was called with values from the following ***sources***:
  - virtualinvoke $r5.<**android.telephony.TelephonyManager: java.lang.String getDeviceId()**>() in method <org.cert.sendsms.Button1Listener: void onClick(android.view.View)>