FlowDroid:Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps.

Author: S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel.

Presented by: Kruti Sharma

CS594: Software Testing, Verification and Validation

Introduction

- Android phones are used in our everyday life and it is very common for the apps to disclose sensitive information to any storage, logs or any advertisers.

Some insights about Sensitive Information

- True Caller

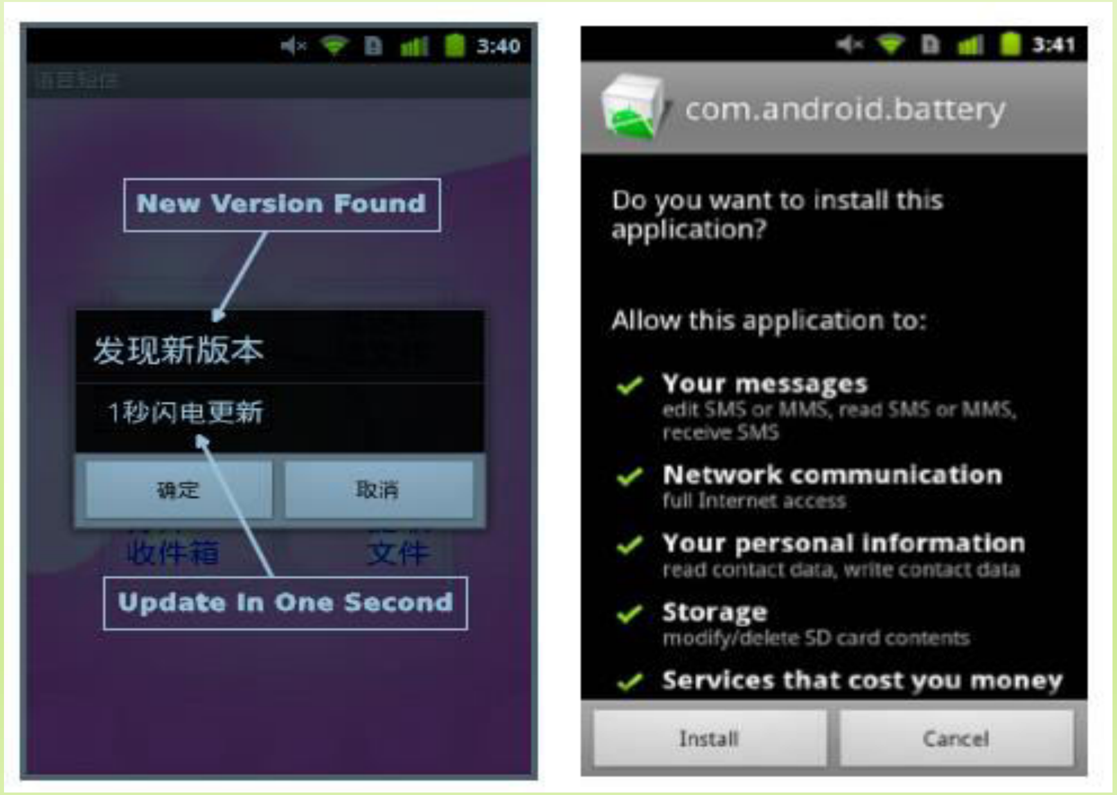
How many of you have used or are using this app ?

Can anyone explain me – how does it functions ?

IMEI is the heart of phone handset. It is mostly used to authenticate in terms of criminal activities to verify whether a call has been made from a specific handset. Some malware applications were even found to receive data through broadcast receivers and to then send out this data in SMS messages. This can allow other applications to send SMS messages indirectly, without requiring the respective permission on their own.

Problem

- Sensitive data disclosures
- Leak private data through a dangerously broad set of permission granted by the user – i.e. allow the application to read/edit your messages, network communication, personal information (read/write contact details etc) etc.


3:40

New Version Found

发现新版本

1秒闪电更新

确定 取消

收件箱 文件

Update In One Second

3:41

com.android.battery

Do you want to install this application?

Allow this application to:

✓ **Your messages**
edit SMS or MMS, read SMS or MMS, receive SMS

✓ **Network communication**
full Internet access

✓ **Your personal information**
read contact data, write contact data

✓ **Storage**
modify/delete SD card contents

✓ **Services that cost you money**

Install Cancel

Motivation

- Detect malicious apps that leak sensitive data.
  - Example: leak contacts list to advertisers or any third party companies.
  - Permission model - broad set of permission granted by the user.
  - Stealing user credentials.
  - As per one of the research conducted on some 30 applications (which access our GPS, camera or microphone) – 15 out of 30 applications shared location information with advertisement servers and some shared phone identifiers with a remote server.

Android Overview

- Multiple entry points – the program does not have any main method.
- Four different components defined by an App Developer:
  - Activities (screens, interacted by users)
  - Services (performs background tasks)
  - Content Providers (database)
  - Broadcast Receivers (listen for global events)
- Asynchronous executing components
- Callbacks

Current Methods

- Scandroid
- Androidleaks
- Chex
- Leakminer

The complexity faced by these methods is due to the lifecycle of the android application i.e.

- Multiple entry points.
- Recognizing different callbacks.
- Asynchronous execution of components.
- Detection of sensitive information may require analysis of manifest and layout XML files.

Taint Analysis

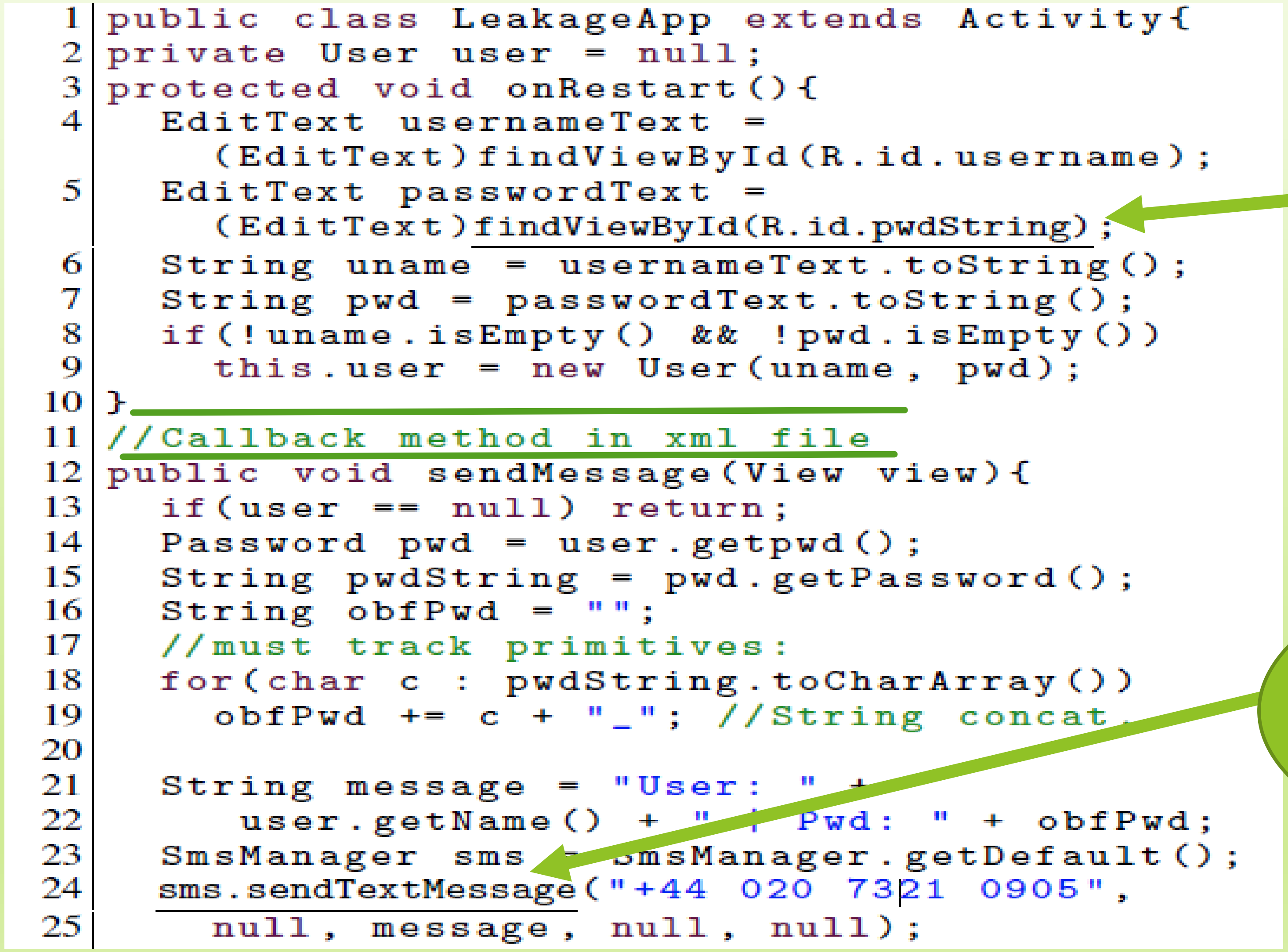
A taint analysis tracks data from predefined data sources (ex- contacts list) to predefined data sinks (ex- internet) and aims at discovering all connections between these sources and sinks

We deal with two types of analysis:

1. Static program analysis - analysis of computer software that is performed without actually executing programs**.**
2. Dynamic program analysis - analysis of computer software that is performed on executing programs (this is important in order to have a proper code coverage.)

Both the approaches track sensitive "tainted" information through the application by starting at a pre-defined Source -> flowing into a Sink – giving information about which data may be leaked.

A sample Android Application



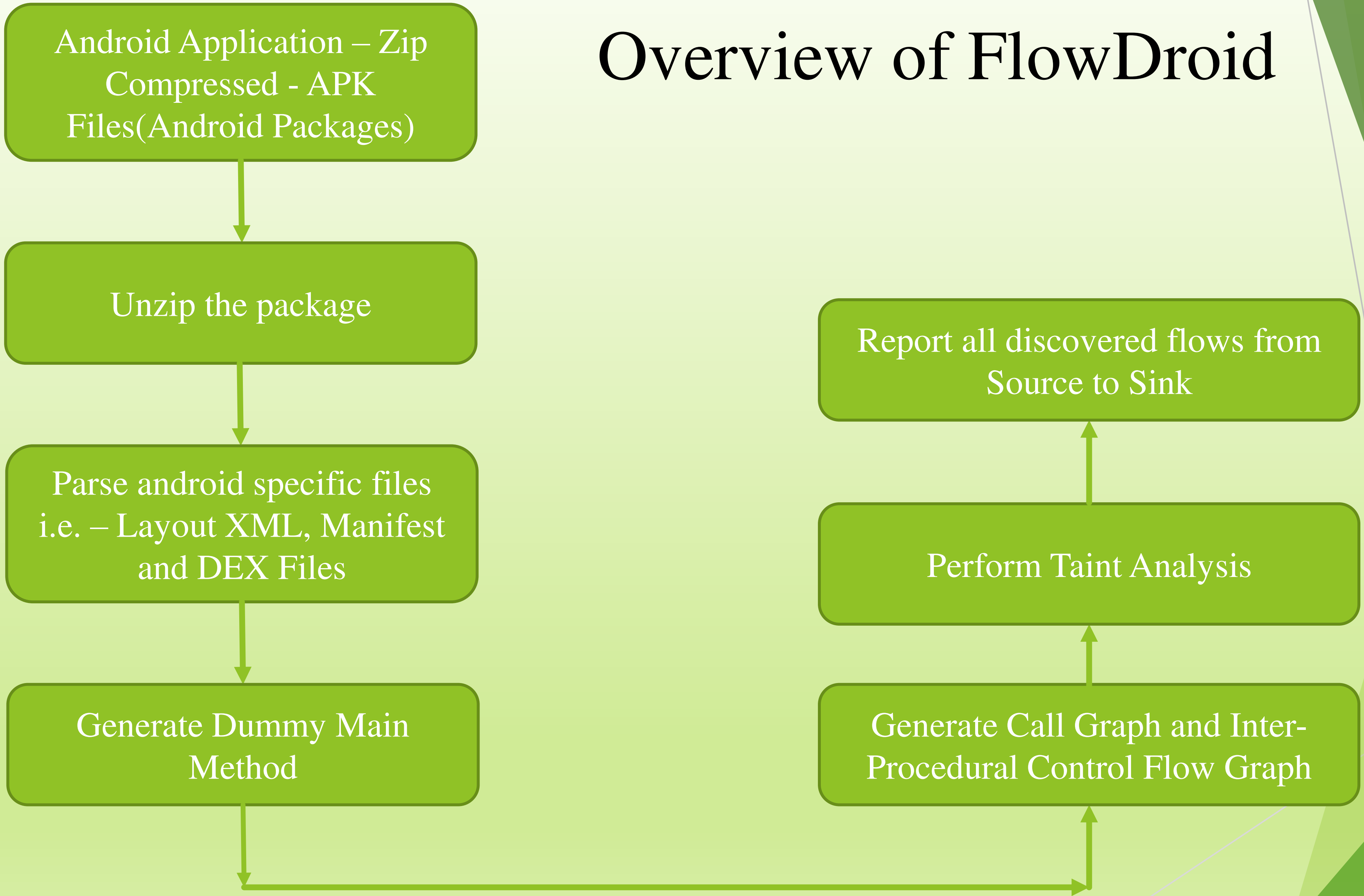
```
public class LeakageApp extends Activity{
private User user = null;
protected void onRestart(){
  EditText usernameText =
    (EditText)findViewById(R.id.username);
  EditText passwordText =
    (EditText)findViewById(R.id.pwdString);
  String uname = usernameText.toString();
  String pwd = passwordText.toString();
  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);
```

The app reads the password field – this is a sensitive information - Source

This method calls another method in order to send the password as a Text Message- Sink

Overview of FlowDroid


Android Application – Zip Compressed - APK Files(Android Packages)
Unzip the package
Parse android specific files i.e. – Layout XML, Manifest and DEX Files
Generate Dummy Main Method
Generate Call Graph and Inter-Procedural Control Flow Graph
Perform Taint Analysis
Report all discovered flows from Source to Sink

Android Example

```java
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);
```

Control Flow Graph

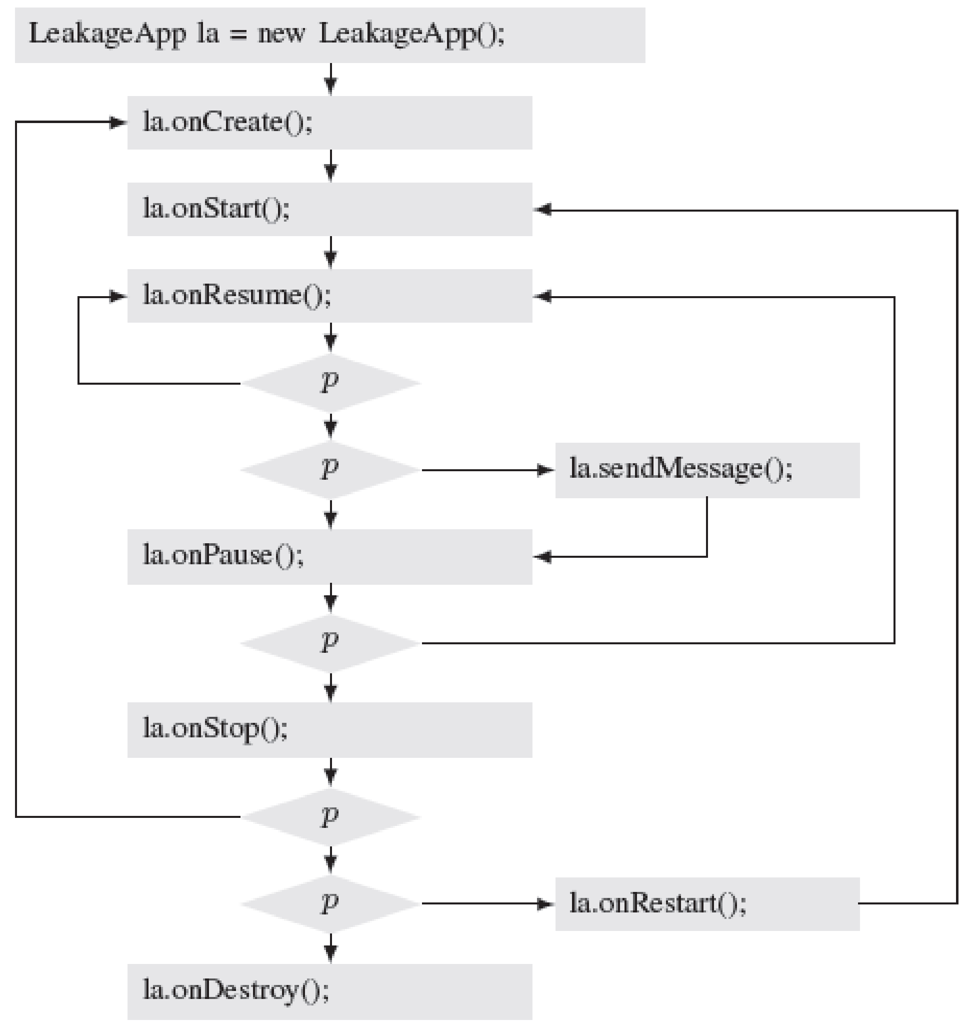


Figure 1: CFG for dummy main method

Precise Flow and Taint Analysis

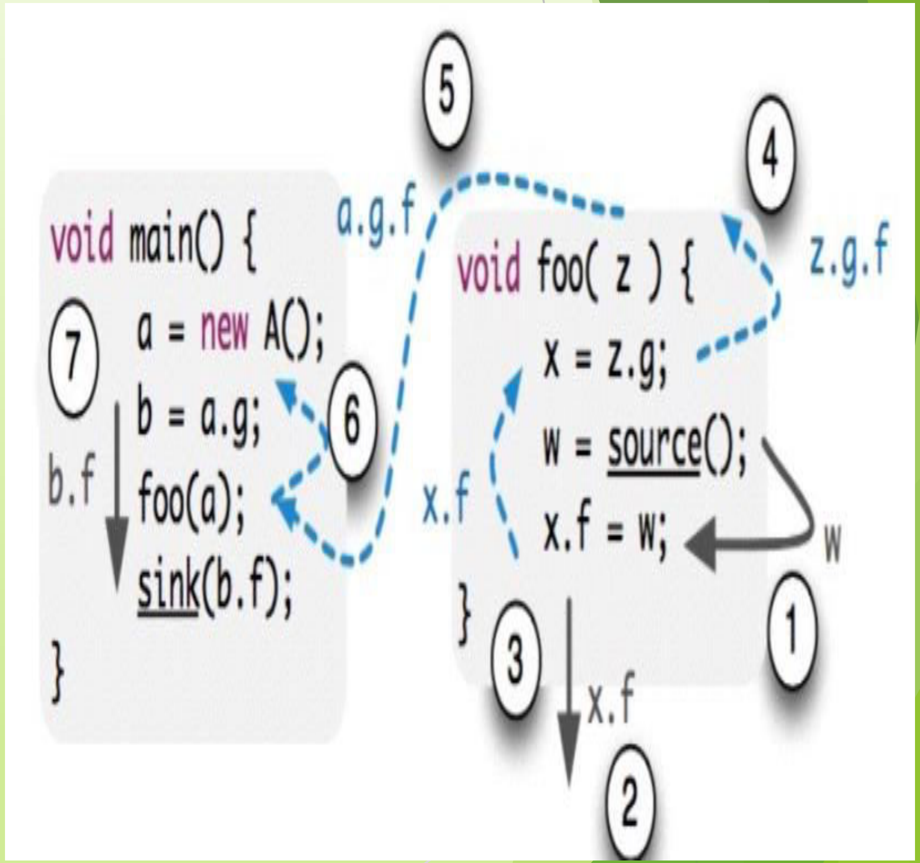
- Access Path:

An Access path implicitly describes the set of all objects reachable through this path.

- Step 1 – tainted variable w is assigned to heap object x.f
- Step 2- Continue taint tracking for w and x.f
- Step 3- Whenever a heap object gets tainted, the backward analysis searches upwards for aliases of the respective object (x.f)

So now we find – b.f = this has been tainted and propagated forward.

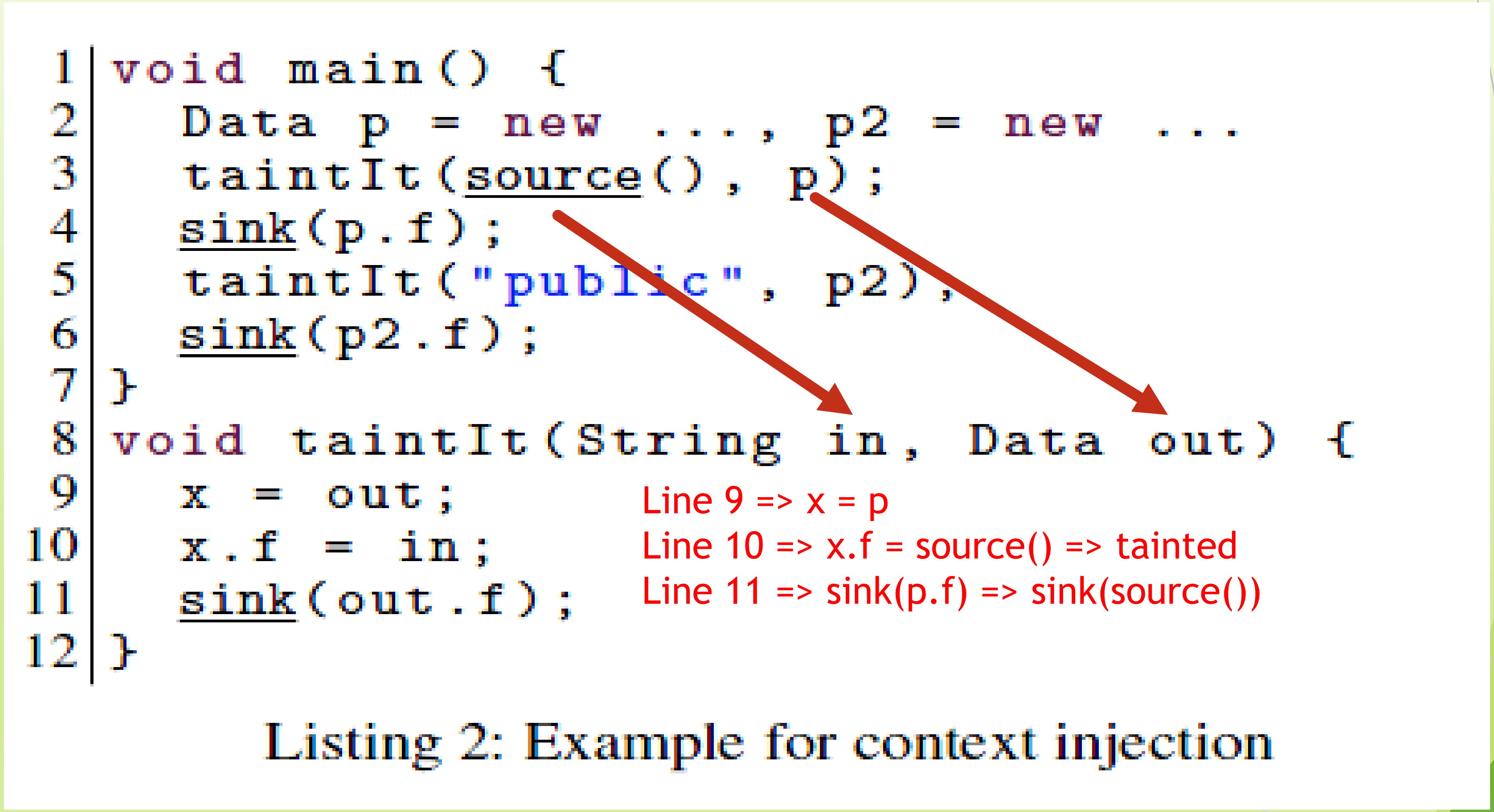
- x.f, x.g.f
- FlowDroid - Configurable lengths (default = 5)



```
void main() {
    a = new A();
    b = a.g;
    foo(a);
    sink(b.f);
}
```

```
void foo( z ) {
    x = z.g;
    w = source();
    x.f = w;
}
```

On Demand Alias Analysis



```
 1 void main() {
 2   Data p = new ..., p2 = new ...
 3   taintIt(source(), p);
 4   sink(p.f);
 5   taintIt("public", p2);
 6   sink(p2.f);
 7 }
 8 void taintIt(String in, Data out) {
 9   x = out;
10   x.f = in;
11   sink(out.f);
12 }
```

Line 9 => x = p
Line 10 => x.f = source() => tainted
Line 11 => sink(p.f) => sink(source())

Listing 2: Example for context injection

Cont. On demand alias analysis

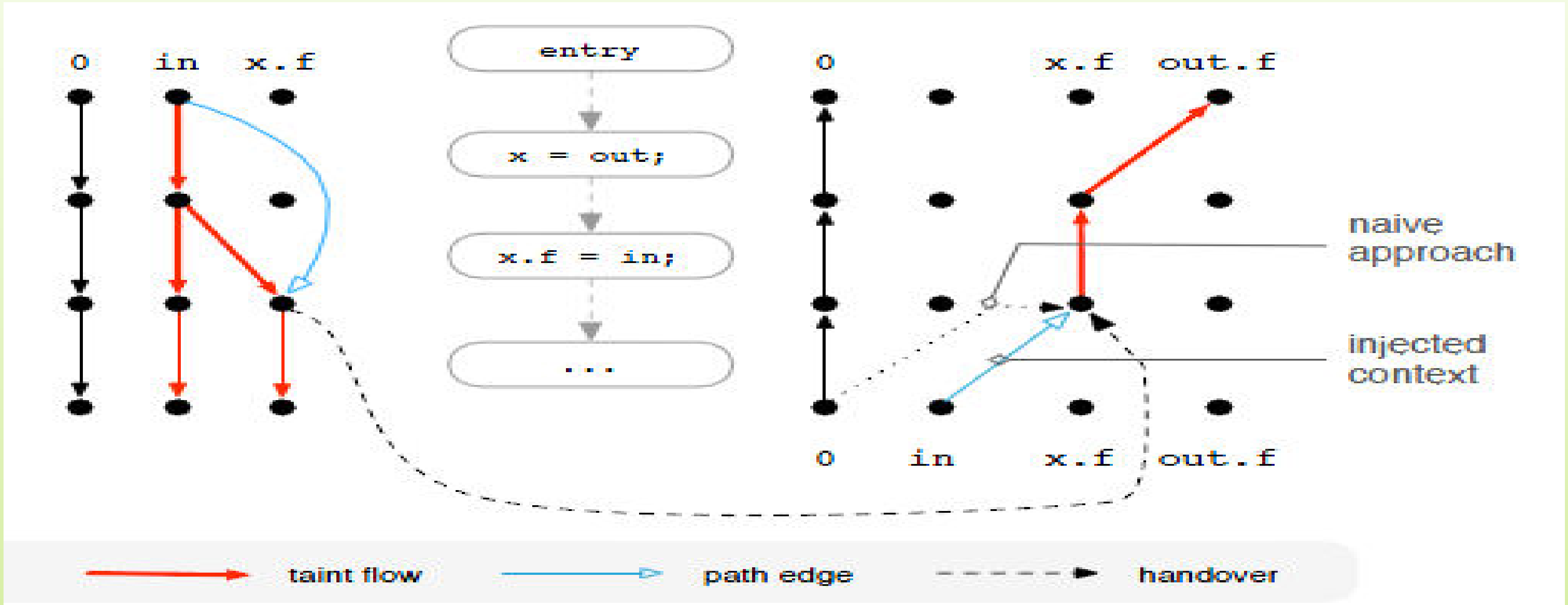
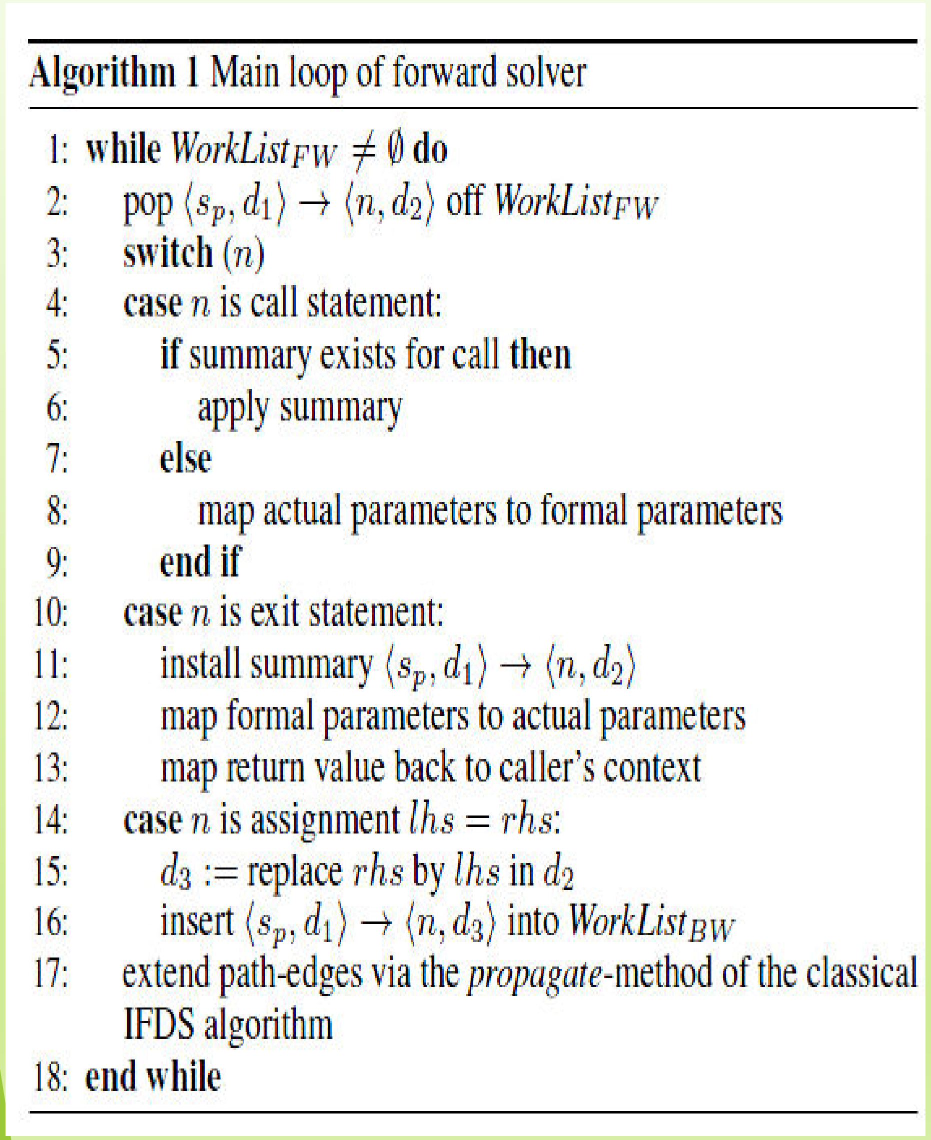


Figure 3: Analysis handover with context injection in taintIt

Inject context(outcome) of Forward Analysis into Backward Analysis => not all inputs leads to taints.

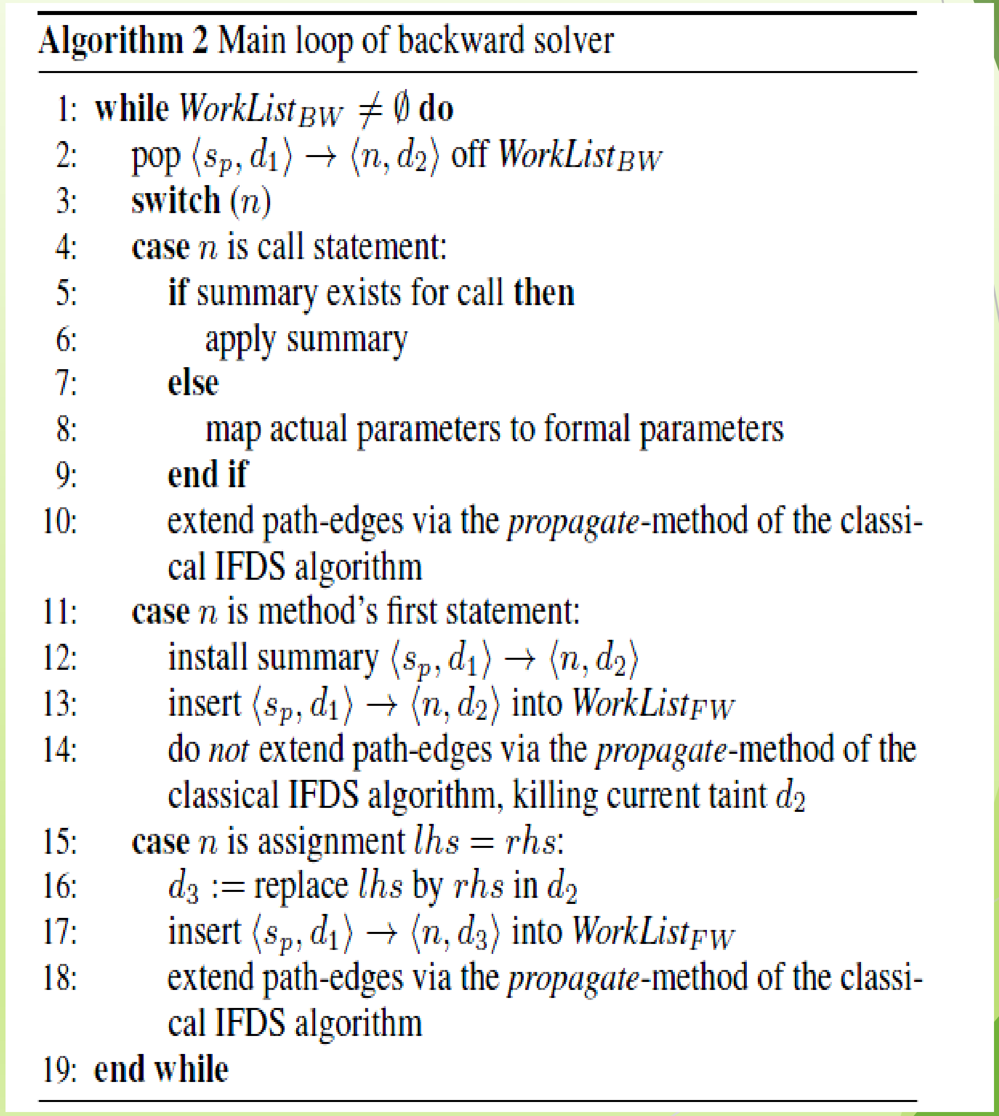
Line 5- tainIt("public", p2) => this normally assigns the heap object with string.

Algorithm - Forward and Backward Solver



Algorithm 1 Main loop of forward solver

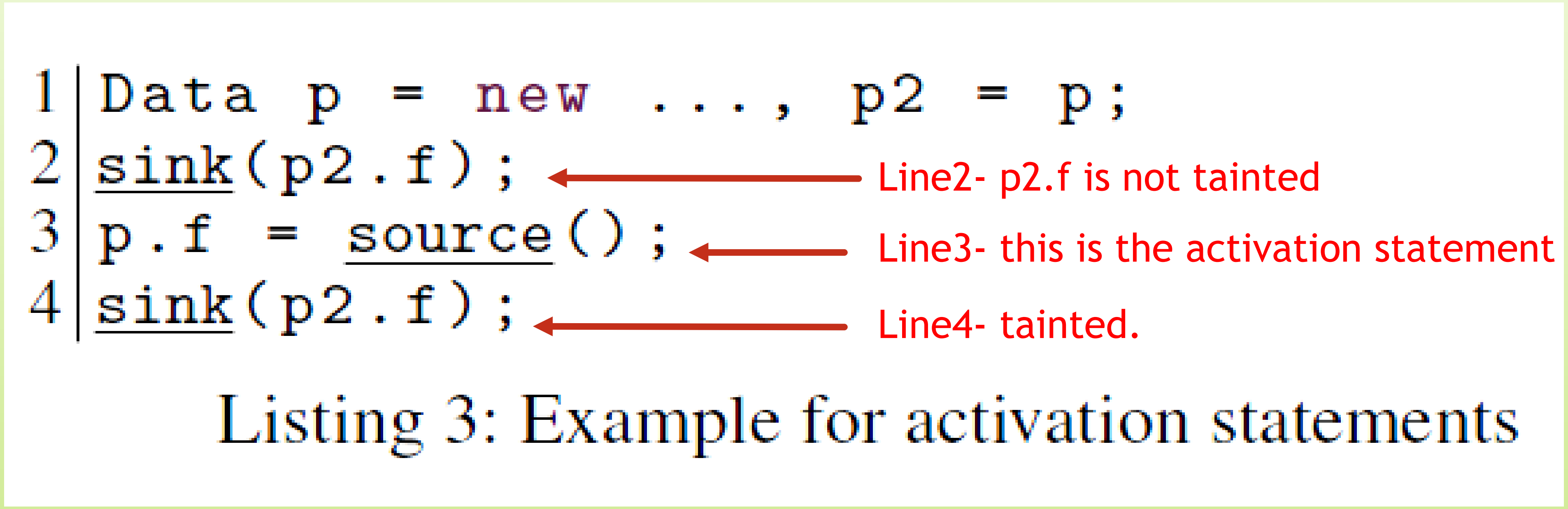
```
1:  while $WorkList_{FW} \neq \emptyset$ do
2:      pop $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ off $WorkList_{FW}$
3:      switch $(n)$
4:      case $n$ is call statement:
5:          if summary exists for call then
6:              apply summary
7:          else
8:              map actual parameters to formal parameters
9:          end if
10:     case $n$ is exit statement:
11:         install summary $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$
12:         map formal parameters to actual parameters
13:         map return value back to caller's context
14:     case $n$ is assignment $lhs = rhs$:
15:         $d_3$ := replace $rhs$ by $lhs$ in $d_2$
16:         insert $\langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle$ into $WorkList_{BW}$
17:     extend path-edges via the propagate-method of the classical
        IFDS algorithm
18: end while
```



Algorithm 2 Main loop of backward solver

```
1:  while $WorkList_{BW} \neq \emptyset$ do
2:      pop $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ off $WorkList_{BW}$
3:      switch $(n)$
4:      case $n$ is call statement:
5:          if summary exists for call then
6:              apply summary
7:          else
8:              map actual parameters to formal parameters
9:          end if
10:         extend path-edges via the propagate-method of the classi-
            cal IFDS algorithm
11:     case $n$ is method's first statement:
12:         install summary $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$
13:         insert $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ into $WorkList_{FW}$
14:         do not extend path-edges via the propagate-method of the
            classical IFDS algorithm, killing current taint $d_2$
15:     case $n$ is assignment $lhs = rhs$:
16:         $d_3$ := replace $lhs$ by $rhs$ in $d_2$
17:         insert $\langle s_p, d_1 \rangle \rightarrow \langle n, d_3 \rangle$ into $WorkList_{FW}$
18:         extend path-edges via the propagate-method of the classi-
            cal IFDS algorithm
19: end while
```

Maintain Flow sensitivity



```
1 Data p = new ..., p2 = p;
2 sink(p2.f);
3 p.f = source();
4 sink(p2.f);
```

Line2- p2.f is not tainted

Line3- this is the activation statement

Line4- tainted.

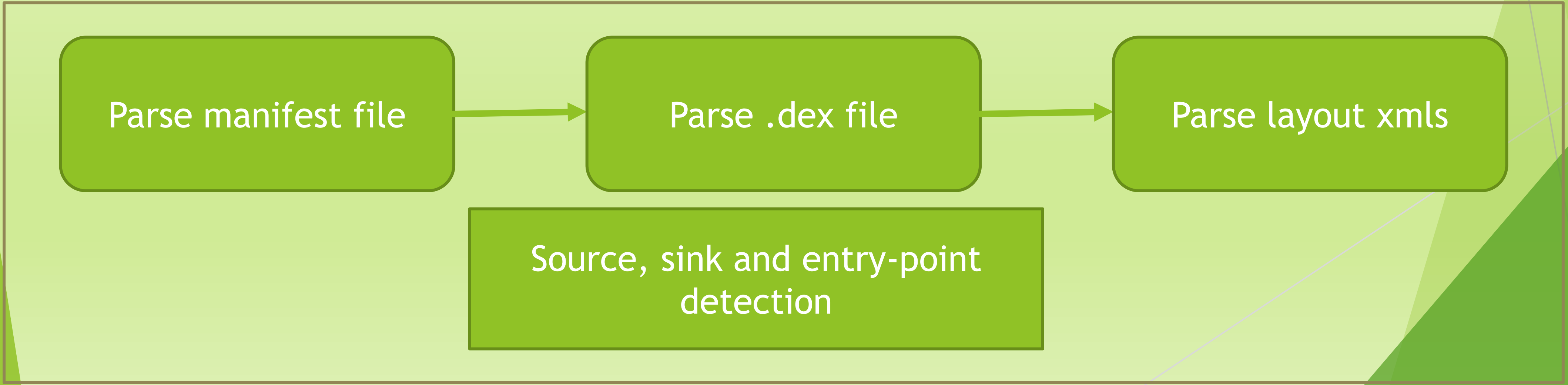
Listing 3: Example for activation statements

FlowDroid Modelling Android Lifecycle

- **Multiple Entry Points:** Creates a dummy main method for android application emulating the lifecycle, this helps to analyze all possible transitions in the Android lifecycle.
- **Asynchronous executing components**: Models the execution assuming that the components can run in any sequence and hence bases its analysis on IFDS Framework - by joining analysis results immediately at any control-flow merge point.
- **Callbacks**: any callbacks for example: location updates are all stored in the dummy main method along with that, the component is associated with the callback they register. FlowDroid computes one call graph per component

FlowDroid Architecture

- FlowDroid extends:
  - Soot Framework – this provides three-address code intermediate representation Jimple
  - Spark Framework – for call graph analysis
- Uses a plugin – Dexpler – convert Androids Dalvik bytecode into Jimple
- Heros Framework – scalable, highly multi-threaded implementation of IFDS Framework.



Experimental Evaluation

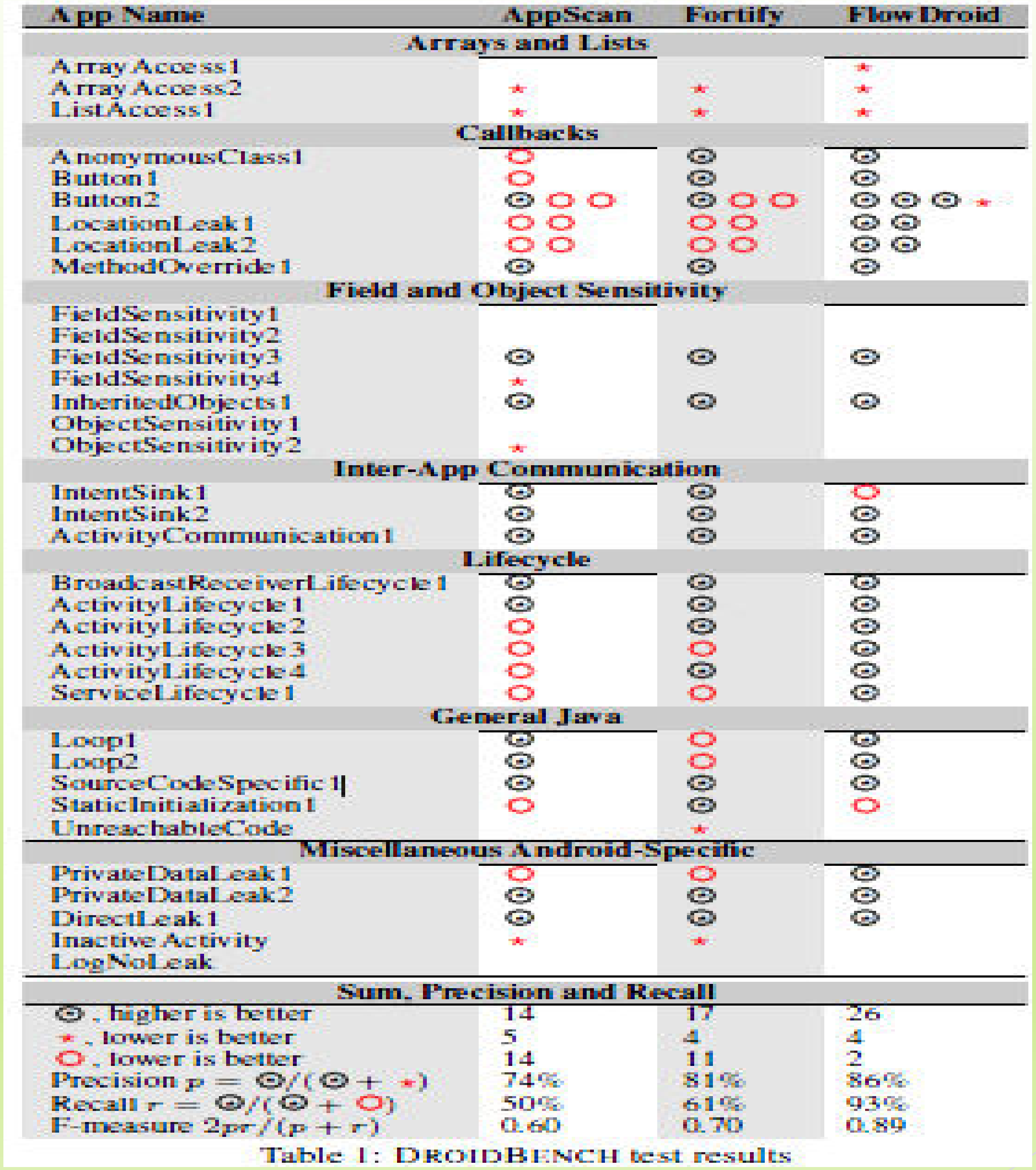
- RQ1 How does FlowDroid compare to commercial taint-analysis tools for Android in terms of precision and recall?

DroidBench –

- Android specific test-suite, specifically developed to handle Android Lifecycle, callbacks or interaction with UI.
- 39 hand-crafted Android apps
- Precision of 86% and recall 93% which is much better as compared to AppScan Source and Fortify SCA.

(Precision = correct warning / (correct warning + false warning))

(Recall = correct warning / (correct warning + missed leak))



App Name	AppScan	Fortify	FlowDroid
Arrays and Lists			
ArrayAccess1			★
ArrayAccess2	★	★	★
ListAccess1	★	★	★
Callbacks			
AnonymousClass1	○	⊙	⊙
Button1	○	⊙	⊙
Button2	⊙ ○ ○	⊙ ○ ○	⊙ ⊙ ⊙ ★
LocationLeak1	○ ○	○ ○	⊙ ⊙
LocationLeak2	○ ○	○ ○	⊙ ⊙
MethodOverride1	⊙	⊙	⊙
Field and Object Sensitivity			
FieldSensitivity1			
FieldSensitivity2			
FieldSensitivity3	⊙	⊙	⊙
FieldSensitivity4	★		
InheritedObjects1	⊙	⊙	⊙
ObjectSensitivity1			
ObjectSensitivity2	★		
Inter-App Communication			
IntentSink1	⊙	⊙	○
IntentSink2	⊙	⊙	⊙
ActivityCommunication1	⊙	⊙	⊙
Lifecycle			
BroadcastReceiverLifecycle1	⊙	⊙	⊙
ActivityLifecycle1	⊙	⊙	⊙
ActivityLifecycle2	○	⊙	⊙
ActivityLifecycle3	○	○	⊙
ActivityLifecycle4	○	⊙	⊙
ServiceLifecycle1	○	○	⊙
General Java			
Loop1	⊙	○	⊙
Loop2	⊙	○	⊙
SourceCodeSpecific1	⊙	⊙	⊙
StaticInitialization1	○	⊙	○
UnreachableCode		★	
Miscellaneous Android-Specific			
PrivateDataLeak1	○	○	⊙
PrivateDataLeak2	⊙	⊙	⊙
DirectLeak1	⊙	⊙	⊙
InactiveActivity	★	★	
LogNoLeak			
Sum, Precision and Recall			
⊙, higher is better	14	17	26
★, lower is better	5	4	4
○, lower is better	14	11	2
Precision $p = ⊙/(⊙ + ★)$	74%	81%	86%
Recall $r = ⊙/(⊙ + ○)$	50%	61%	93%
F-measure $2pr/(p + r)$	0.60	0.70	0.89

Table 1: DroidBench test results

Cont. Experimental Evaluation

- **RQ2: Performance on InsecureBank:**

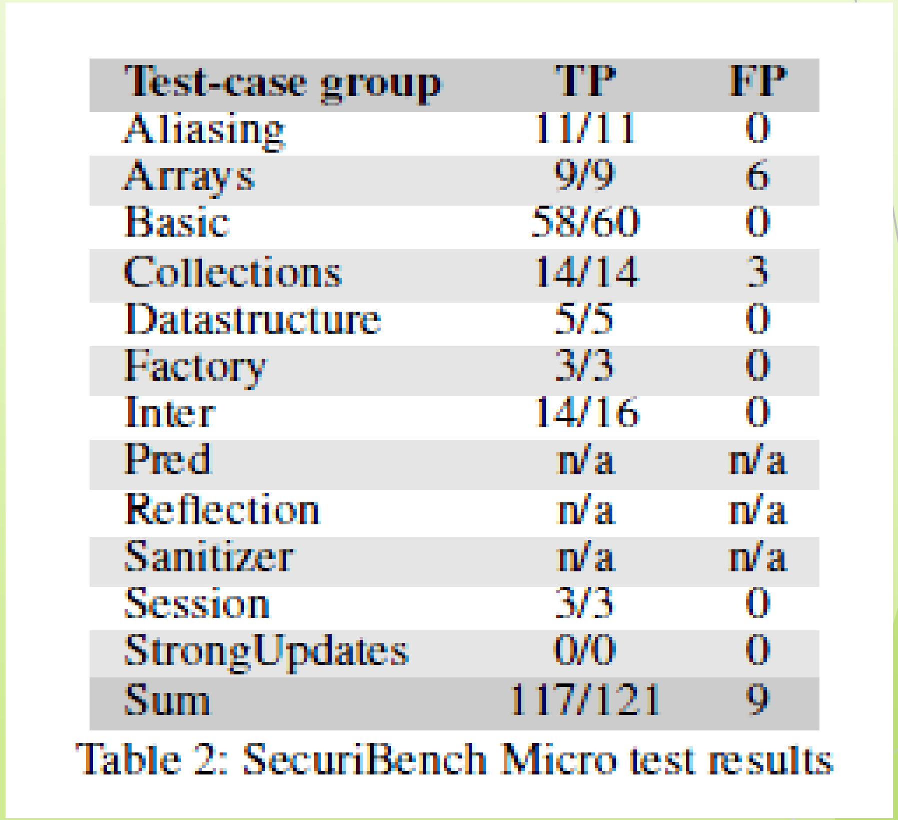
InsecureBank is basically a vulnerable App designed to test analysis tools.

  - Analysis of App: 31 seconds
  - Detects all 7 data leaks.
  - No false positives or no false negatives

- **RQ3: Performance on Real-World Applications:**
  - Ran FlowDroid on 500 Google Play apps = no leaks.
  - Again ran on 1000 known malware samples from VirusShare project = average 2 data leaks.

Cont. Experimental Evaluation

- **<u>RQ4: SecuriBench Micro:</u>** intended for web-based applications
  - The number of actual leaks reported (117/121) and false positives (9) gives good results for FlowDroid.



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

Table 2: SecuriBench Micro test results

Summarizing FlowDroid

1. Novel static taint-analysis system for Android – analyses both app byte-code and configuration files.
2. First context, field, object and flow- sensitive taint analysis i.e. –
    - On-demand alias supports the context and object sensitivities – analyzing the information flows lazily (demand driven) instead of eagerly computing all the data-flows.
    - Field analysis- every field may not contain a sensitive information, so analyzing such fields in flow is not mandatory.
3. Detect data flows – whether caused by carelessness or malicious intent.

Limitations

- Resolves reflective calls only if their arguments are string constants.
- Handles arrays imprecisely
- Assumption – threads execute in any arbitrary but sequential order and thus does not accounts for multiple threads.

References

- http://dl.acm.org/citation.cfm?id=2594299
- http://www.ieee-security.org/TC/SP2012/papers/4681a095.pdf
- http://people.cs.vt.edu/~ryder/6304/lectures/10-ArztEtAl-FlowDroid-PLDI2014-DanBarton.pdf
- https://courses.cs.washington.edu/courses/cse501/15sp/slides/L8-phone-malware1.pdf
- https://ec-spride.sit.fraunhofer.de/fileadmin/user_upload/Group_EC_Spride/FinishedTheses/Masterthesis20130702.pdf
- http://security.stackexchange.com/questions/49343/what-is-the-risk-of-leaking-imei-imsi-numbers-over-a-network

THANK YOU !!!!