RunDroid:

Recovering Execution Call Graphs for Android Applications

Yujie Yuan[1] Lihua Xu [1] Xusheng Xiao[2]
Andy Podgurski[2] Huibiao Zhu[1]

[1]East China Normal University, China

[2]Case Western Reserve University, USA

ESEC/FSE 2017, SEPTEMBER 04 - 08

Table of Contents

Why we need RunDroid

Intention

Tell us what happen during app running?

Challenges in Android

- Implicit callbacks
- Lifecycle methods
- Multi-thread communications

Analysis tools in Android

- Static Analysis $\longrightarrow$ time & space cost
- Dynamic Analysis $\longrightarrow$ time & technique cost

Basic idea in RunDroid

RunDroid

A tool that captures the dynamic method executions during each app running, and recovers the complete dynamic call graph to help people know what happen during app running.

Overview

RunDroid takes the source code of an app as input, instruments the source code, and intercepts the executions of the instrumented app to analyze message objects. After each execution, RunDroid produces a set of log files, which will be further analyzed to generate the dynamic call graph for the execution.

Note

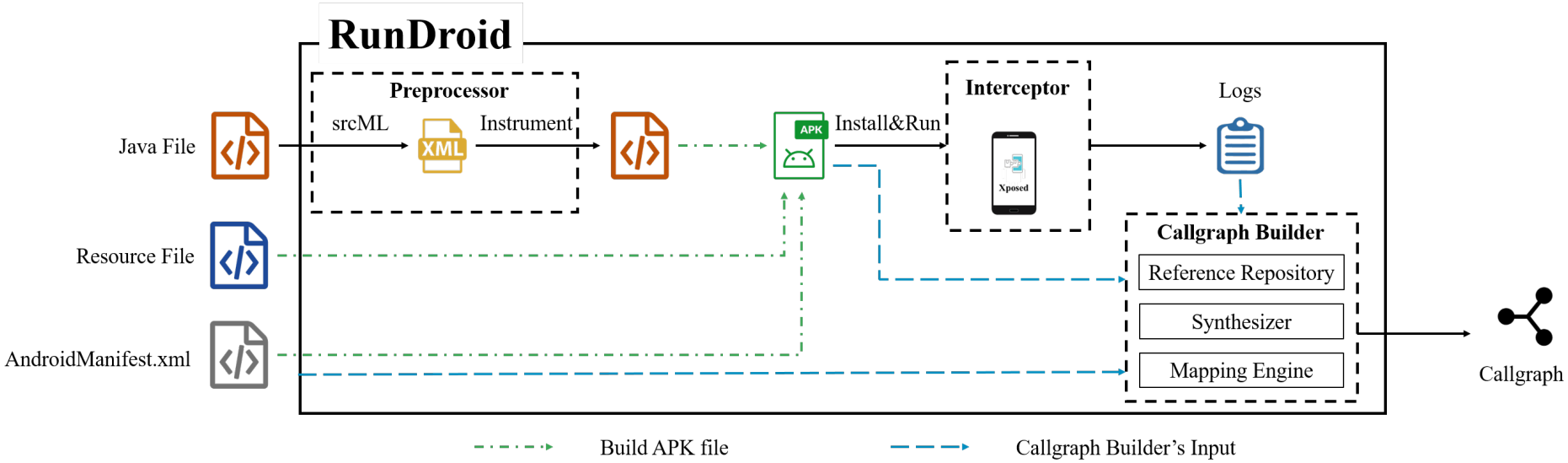
We assume we can access the code of programs.

RunDroid's Framework

Three steps

Three steps:

1. Capture application layer method calls
2. Recover method calls between app and the Android framework
3. Build dynamic call graphs



How RunDroid works

Step 1:Capture application layer method calls[1]

Basic idea:

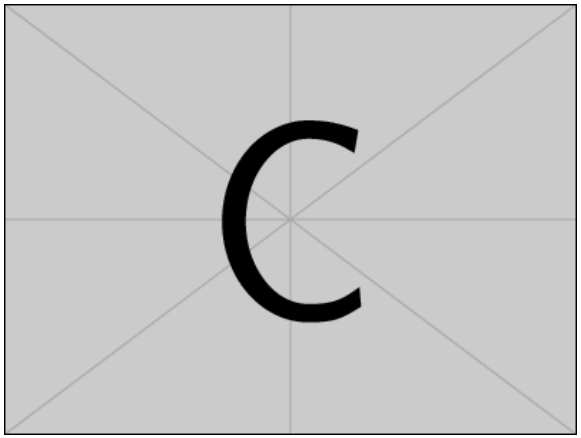
Log the target method's information before and after method's executed.

Challenges:

The 64K reference limit(Emma) $\Longrightarrow$ Instrument on source code.

Process:

Java file $\longrightarrow$ xml File $\longrightarrow$ xml file after instrumented $\longrightarrow$ The code with log methods.



[1] Application Layer Method Calls:the method defined by user/developer.

How RunDroid works

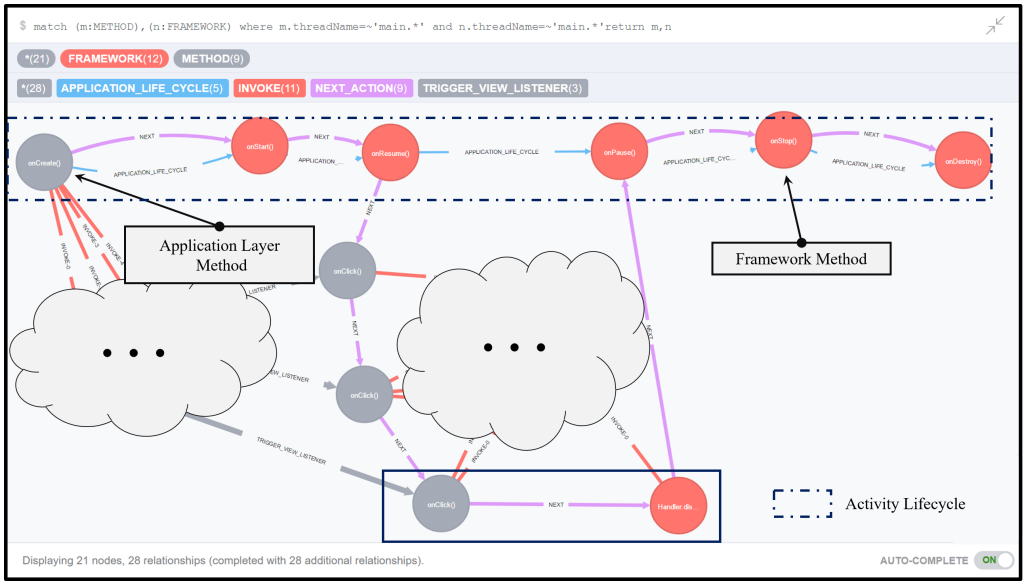
Step 2: Recover method calls between app and the Android framework

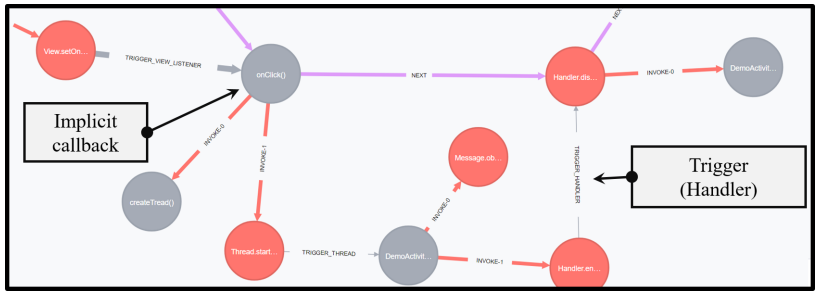
How RunDroid works

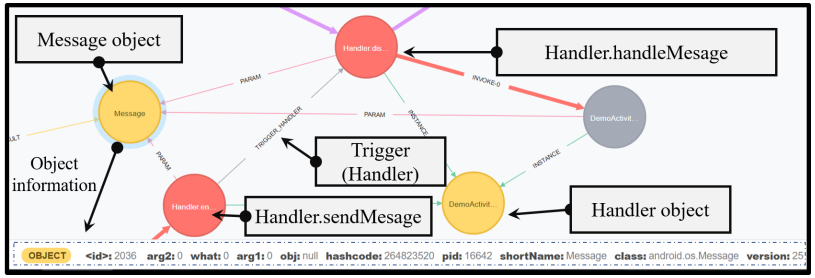
Step 3: Build dynamic call graphs

RunDroid for Visualization

- **The exact event sequences**, instead of all possible ones as the static analyzers do;
- **The execution call graphs**, that are typically hard to capture with static analyzers;
- **Related object information**, that assist in data flow analysis of static analyzers







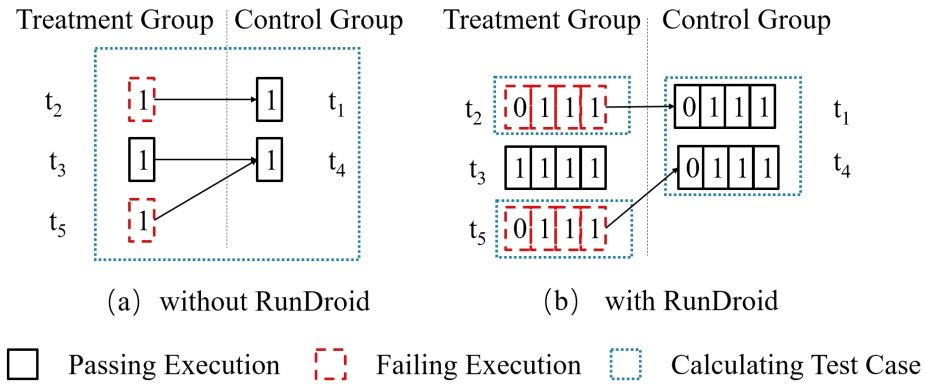
RunDroid on fault localization

To illustrate how the dynamic call graphs built by RunDroid assists fault localization techniques, we compare the estimation results using **the causal influence model** with or without RunDroid

```
1   void onClick(View v) {
2     num = getNumber();
3     if(v.getId() == R.id.btn1) {
4       if( num == 0 ) {
5         num=1;
6       }
7     }
8     Thread t = createThread(v.getId());
9     t.start();
10  }
11  TaskThread.run() {
12    if(v.getId() == R.id.btn1) {
13      loadData(num); /* FAULT */
14    }
15  }
```

RunDroid on fault localization

Focs on Line 13:




```
1   void onClick(View v) {
2     num = getNumber();
3     if(v.getId() == R.id.btn1) {
4       if( num == 0 ) {
5         num=1;
6       }
7     }
8     Thread t = createThread(v.getId());
9     t.start();
10  }
11  TaskThread.run() {
12    if(v.getId() == R.id.btn1) {
13      loadData(num); /* FAULT */
14    }
15  }
```

RunDroid on fault localization

Table: Comparing Results

		t_1	t_2	t_3	t_4	t_5	τ	τ'
1	void onClick(View v) {							
2	num = getNumber();	1	1	1	1	1	NA	NA
3	if(v.getId() == R.id.btn1) {	1	1	1	1	1	NA	NA
4	if(num == 0) {	0	1	1	0	1	0.67	0.67
5	num=1;	0	0	1	0	0	-1.0	-1
6	}							
7	}							
8	Thread t = createThread(v.getId());	1	1	1	1	1	NA	NA
9	t.start();	1	1	1	1	1	0.67	0.67
10	}							
11	TaskThread.run() {							
12	if(v.getId() == R.id.btn1) {	1	1	1	1	1	NA	0.67
13	**loadData(num); /* FAULT */**	0	1	1	0	1	0.67	1
14	}							
15	}							
		0	1	0	0	1		

Customized work & Furture work

Customized work

- customize the log output about object information
- add or remove Android framework methods to hook the methods' executions
- add or remove more method-trigger relationships in Call Graph

Furture work

- add control flow unit into the call graph