Dynamic Slicing for Android

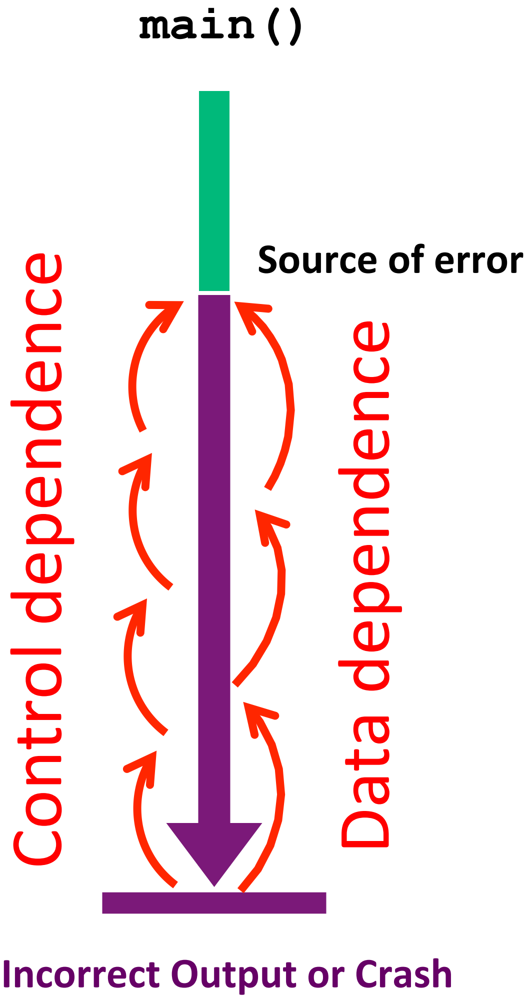
Arash Alavi
Tanzirul Azim
Rajiv Gupta

UC Riverside, University of California

Iulian Neamtiu

NJIT, New Jersey Institute of Technology

"Traditional" Program Slicing



- Incorrect Output or Crash: how was the erroneous value generated?

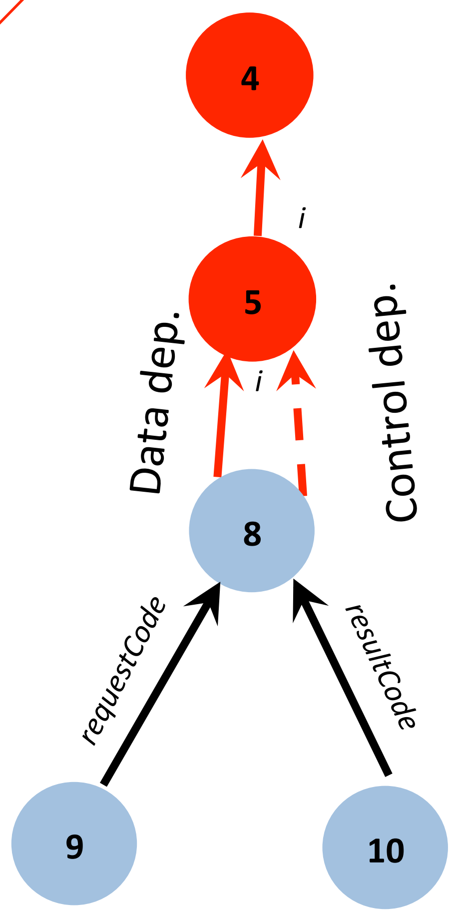
- Backwards Dynamic Slice
  - Subset of executed statements that computed the value
  - Backwards transitive closure over *dynamic data & control dependences*
  - Requires tracing the execution backwards
  - Reduces #instructions to be analyzed drastically (orders of magnitude)

Inter-app communication

```
1  public class GetContacts extends Activity {
2    @Override
3     public void onCreate(Bundle savedInstanceState) {
4       Intent i = new Intent(Intent.ACTION_PICK, Uri.parse("content://contacts"));
5       startActivityForResult(i, PICK_CONTACT_REQUEST);
6    }
7    @Override
8    public void onActivityResult(int requestCode, int resultCode, Intent data) {
9       if (requestCode == PICK_CONTACT_REQUEST) {
10        if (resultCode == RESULT_OK) {
```

no control flow

Another process
(Contacts app)



Solution: Asynchronous dependences

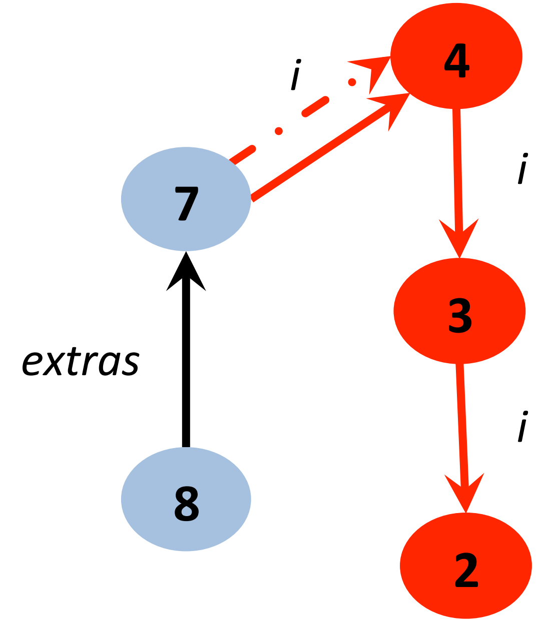
Intra-app communication

```
1 public class ActivityOne extends Activity {...
2   Intent i = new Intent(this, ActivityTwo.class);
3    i.putExtra("Value", "Some Value");
4   startActivity(i);
 5  ...}

 6  public class ActivityTwo extends Activity {...
 7   Bundle extras = getIntent().getExtras();
 8   String value = extras.getString("Value");
 9  ...}
```

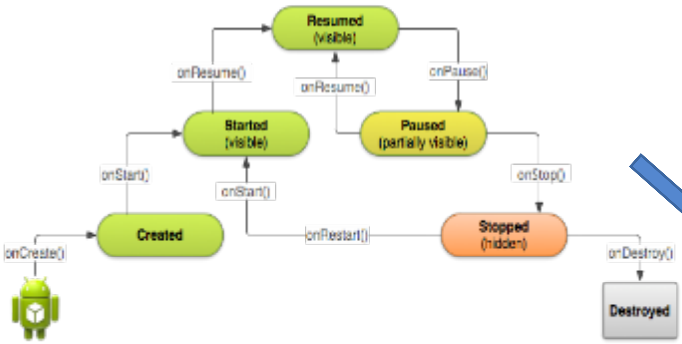
no control flow

Android Framework

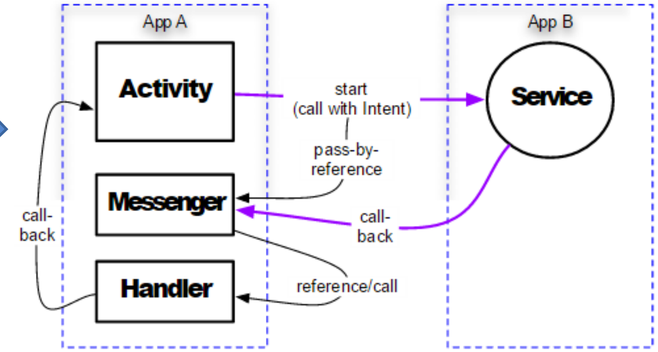


Solution: Track callbacks and AF boundary methods

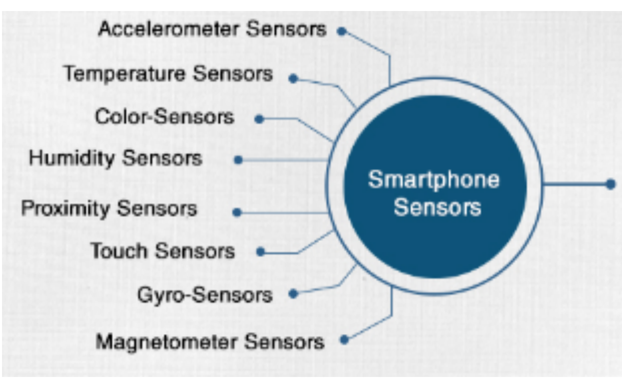
Slicing mobile apps is challenging!



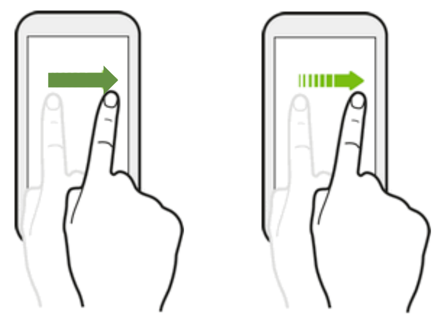
Asynchronous callbacks
- no main()
- **establishing causality??**



Inter- & intra- process communication



High-throughput concurrent sensors



Timing-sensitive

Key novelty: Asynchronous Dependence

Asynchronous Data Dependence (track intents)	Asynchronous Control Dependence (what caused this event)

Asynchronous Data Dependence (track intents)

$$\frac{N_1 = \langle c_1, t_1,, a_1, \ldots \rangle, N_2 = \langle c_2, t_2,, a_1, \ldots \rangle \quad v_2 \in param(c_2), v_1 \in Def(c_1), ref(v_1, t_1) = ref(v_2, t_2)}{N_2 \leftarrow_d N_1 \qquad S_{2t} \leftarrow_d S_{1t}}$$

Inter-app, intra-app, sensors ✔

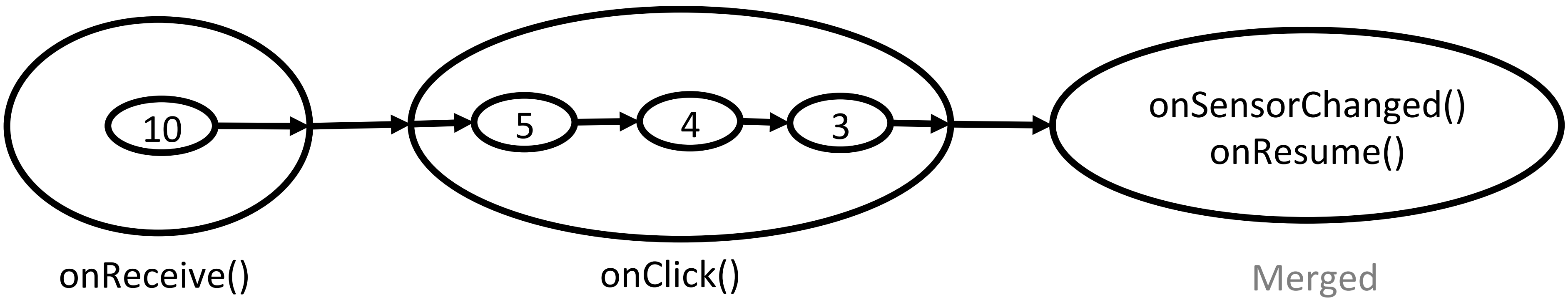
Asynchronous Control Dependence (what caused this event)

$$\frac{N_1 = \langle c_1, t_1, a_1, \ldots \rangle, N_2 = \langle c_2, t_2, a_2, \ldots \rangle, a_1 \neq a_2, initiator(N_2) = N_1, \neg(N_2 \leftarrow_d N_1)}{N_2 \leftarrow_c N_1}$$

$$\frac{N_1 = \langle c_1, t_1, a_1, \ldots \rangle, N_2 = \langle c_2, t_2, a_2, \ldots \rangle, a_1 \neq a_2, initiator(N_2) = N_1, N_2 \leftarrow_d N_1, N_1 \leftarrow_c N_0}{N_2 \leftarrow_c N_0}$$

Causality ✔

Asynchronous dependences: ***Supernodes N*** connected by ***superedges N_1 -> N_2***



Optimizations: Node merging, Loop folding

Sample Program Dependence Graph

Evaluation and Applications

- **Core slicing**
  - 60 apps from a variety of Google Play categories with various bytecode sizes, including the 50MB Twitter behemoth
  - Reduce #instructions to be analyzed from **4,674** to **24**; overhead: **3%**
- **Fault Localization Analysis**
  - Identify the location of a fault in an app
  - 8 real bugs in apps including Notepad (10m+ installs), SoundCloud (100m+ installs), NPR News (1m+ installs)
  - Reduce #instructions from **12,679** to **32**
- **Failure-inducing Input Analysis**
  - Find the input parts responsible for a crash or error
  - Reduce #instructions from **29,267** to **25**
  - 6 real bugs in apps including Etsy (10m+ installs), K-9 Mail (5m+ installs)
- **Minimize Regression Testing**
  - Given two app versions, reduce size of regression test suite from **200** to **8**

Conclusions and Future Work

- **AndroidSlicer**
  - First approach for Android dynamic slicing
  - Effective, efficient, three testing/debugging applications

- **Ongoing/future work:** “needle in a haystack” or dynamic state separation problems
  - **Dynamic taint analysis**
    - Effectiveness (reduce “taint explosion”), efficiency (reduce overhead), implicit flows
  - **Malware analysis**
    - C&C commands -> botnet actions, anti-detection
  - **Undo computing**
    - Restore benign state while eliminating “bad” state
  - **Debugging Deep Learning**
    - Bugs in Code vs Training Datasets: Whom to blame for the Tesla Autopilot crash?