FSE 2016: ACM SIGSOFT International Symposium on the Foundations of Software Engineering

Understanding and Detecting Wake Lock Misuses for Android Applications

Artifact Evaluated




Yepang Liu, Chang Xu, **Shing-Chi Cheung**, and Valerio Terragni

Code Analysis, Testing and Learning Research Group (CASTLE)

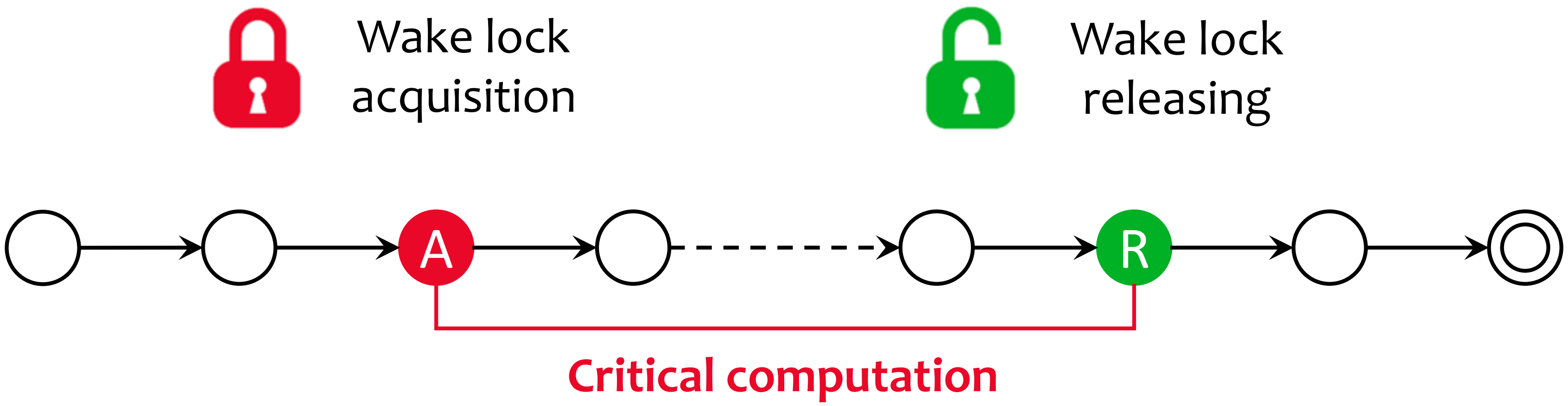
香港科技大學 THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

CASTLE

南京大學 NANJING UNIVERSITY

Wake Lock: Android's Power Control Mechanism

- To save battery power, Android devices quickly fall asleep after a short period of user inactivity
- Wake locks can be used to keep certain hardware (e.g., CPU, Screen) on for **long-running** and **critical** computation (e.g., uninterruptable tasks)



Motivation

- Wake locks can help provide functionalities in a reliable manner
  - **27.2%** apps on Google Play store use wake locks
- However, programming wake locks is non-trivial
  - Various lock types, configuration parameters (e.g., flags)
  - Impact on hardware status and energy consumption

various lock types

Type	CPU	Screen	Keyboard
Partial wake lock	On	Off	Off
Screen dim wake lock	On	Dim	Off
Screen bright wake lock	On	Bright	Off
Full wake lock	On	Bright	Bright
Proximity screen off wake lock	Screen off when proximity sensor activates		

multiple flags: ACQUIRE_CAUSES_WAKEUP, ON_AFTER_RELEASE // flags can be combined

Motivation

Resource errors:

- Unnecessary acquisition
- Leakage
- Permission

Inappropriate use of wake locks is common

- **61.3%** of our investigated open-source apps suffered from various wake lock bugs that can cause app crash, energy waste etc.

- Well capture wake lock misuses?
- Effectively detect them?
- Limitations?
- Automated detection?

wake lock misuses

Research Goals

- Understand the common practices of wake lock usage
- Uncover the common misuses of wake locks
- Design techniques to detect wake lock misuses

Empirical Study: Research Questions

- **Critical computation:** What computational tasks are often protected by wake locks?
- **Wake lock misuses:** Are there common causes of wake lock misuses? What consequences can they cause?

Note: More RQs in our paper and technical report

Two Datasets

1. Binaries (APK files) of **44,736** free Android apps that use wake locks
   - **Comprehensive:** covering all 26 app categories, each category has thousands of apps
   - **Popularity:** each app received 200K+ downloads on average
   - **Diverse sizes:** ranging from a few KB to hundreds of MB, average size 7.7 MB

Study Methodology (Dataset 1)

Decompilation
(Dex2jar)

- Retargeting APK files to Java bytecode

Program analysis
(Soot, BCEL)

- Locating analysis entry points (event handlers)
- Analyzing app API usage, lock type, acquisition/releasing points

Statistical analysis

- Correlating API calls with wake lock uses
- Analyzing common lock types and acquiring/releasing points

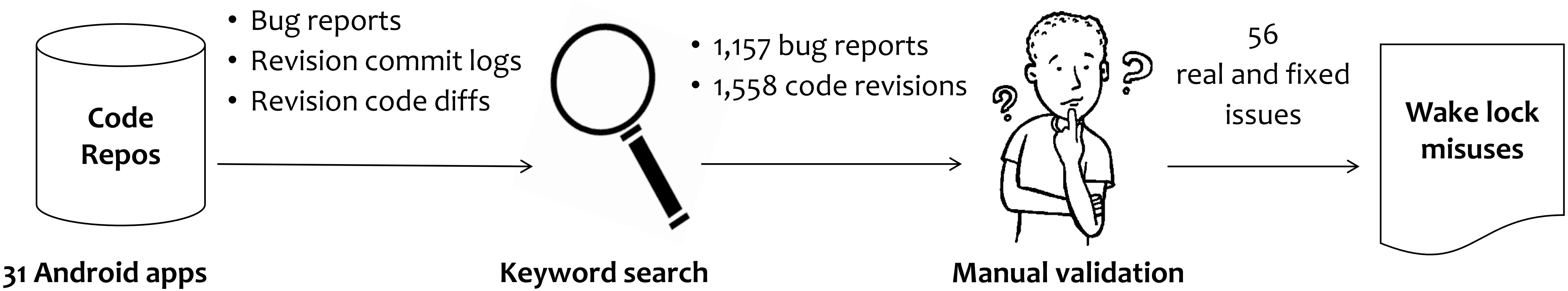
Two Datasets

1. Binaries (APK files) of **44,736** free Android apps that use wake locks
   - **Comprehensive:** covering all 26 app categories, each category has thousands of apps
   - **Popularity:** each app received 200K+ downloads on average
   - **Diverse sizes:** ranging from a few KB to hundreds of MB, average size 7.7 MB

2. Code repositories of **31 most popular** F-Droid indexed **open-source** Android apps that use wake locks
   - **Popularity:** each app received 39+ millions of downloads on average
   - **Well-maintained:** thousands of code revisions, hundreds of bug reports
   - **Large-scale:** each app has 40.3K lines of code on average

Study Methodology (Dataset 2)

- Processing code repositories: search-assisted manual analysis
- Search keywords: wake, wakelock, power, powermanager



Key Empirical Findings (Dataset 1)

- The use of wake locks are strongly correlated with the invocations to APIs that perform 13 types of computational tasks
- Many tasks require permissions to run and can bring users perceptible benefits

Computational task	API example
Networking & communications	java.net.DatagramSocket.connect()
Data management & sharing	android.database.sqlite.SQLiteDatabase.query()
System-level operations	android.os.Process.killProcess()
Media & audio	android.media.AudioTrack.write()
Sensing operations	android.location.LocationManager.requestLocationUpdates()
…	…

Key Empirical Findings (Dataset 2)

✓ Studied by existing work ⊗ Not studied by existing work

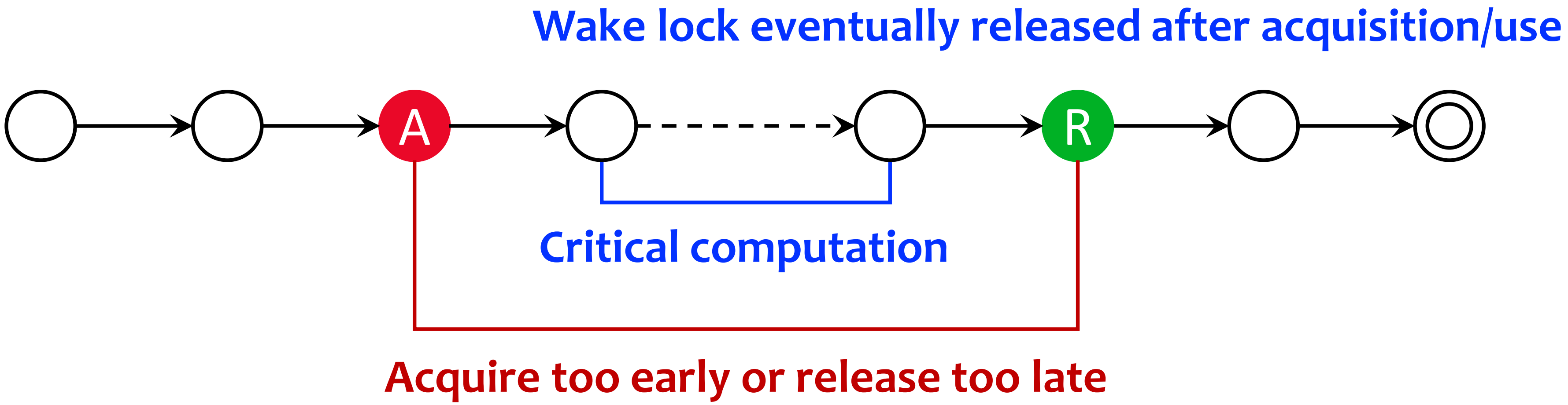
- 8 types of wake lock misuses commonly cause functional/nonfunctional issues

	Root cause	# issues	# affected apps	Example	Consequence
✓	Unnecessary wakeup	11	7	Tomahawk Rev. 883d210	Energy waste
✓	Wake lock leakage	10	7	MyTracks Rev. 1349	Energy waste
⊗	Premature lock releasing	9	7	ConnectBot Issue 37	Crash
⊗	Multiple lock acquisition	8	3	CSipSimple Issue 152	Crash
⊗	Inappropriate lock type	8	3	Osmand Issue 582	Energy waste
⊗	Problematic timeout setting	3	2	K9Mail Issue 170	Instability
⊗	Inappropriate flags	2	2	FBReader Rev. f289863	Energy waste
✓	Permission errors	2	2	Firefox Issue 703661	Crash
	Total	*53*	*18*		

Note: More findings in our paper and technical report

Unnecessary Wakeup

A Wake lock acquisition   R Wake lock releasing



Energy waste

<u>TomaHawk Player bug:</u>

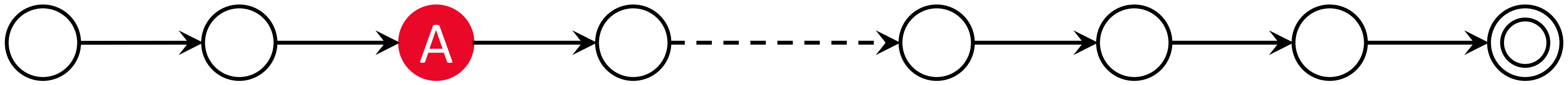
*Wake lock is **not released until** users left the player UI, **should be released immediately after** music stops playing.*

Wake Lock Leakage

A Wake lock acquisition

R Wake lock releasing

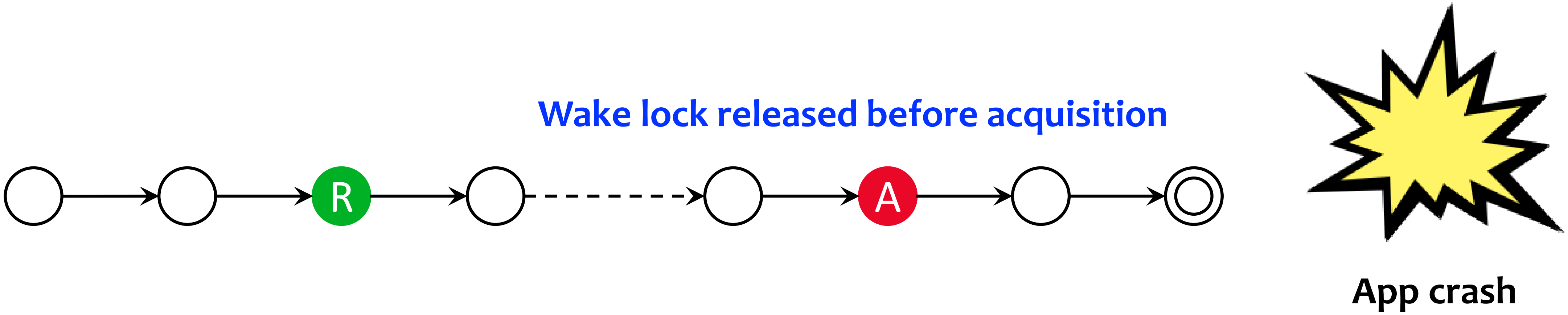
Wake lock never release after use



Energy waste

Premature Lock Releasing

A Wake lock acquisition   R Wake lock releasing



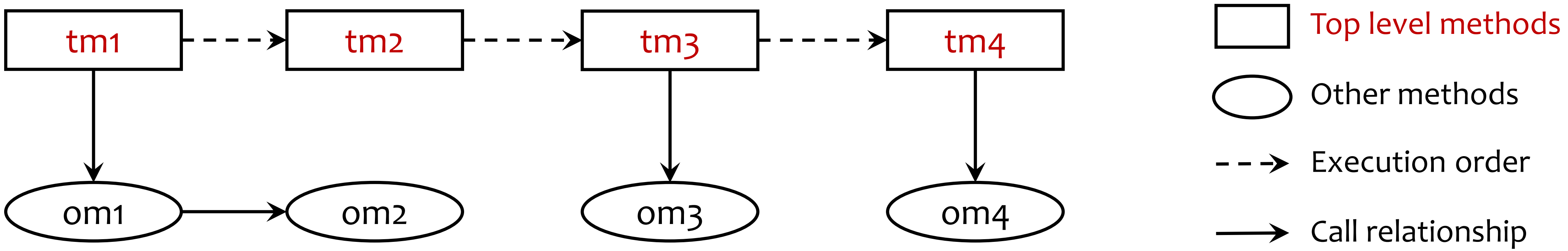
Detecting Wake Lock Misuses

- ELITE: A static wake lock necessity analyzer
- Current version detects (1) unnecessary wakeup and (2) wake lock leakage

Android app | Static analyzer | Wake lock misuses

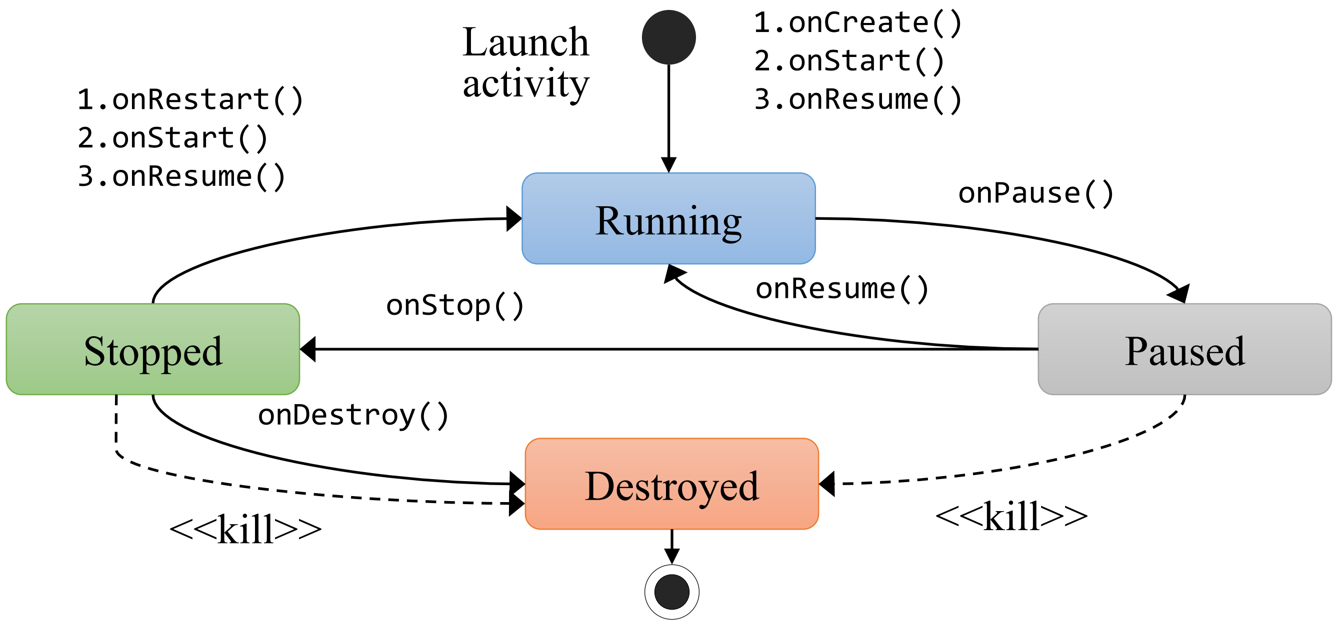
Component-Based Static Analysis

- ELITE analyzes app components one by one. It generates and analyzes "top level method" call sequences for issue detection when analyzing each component.
- **Top level methods:** (1) callback methods and (2) non-callback methods exposed for other components to invoke
- **Execution model:** top level methods (*tm*) are invoked by system or other components and they may invoke various other methods (*om*)



Challenge: Generating Valid Method Call Sequences

- Example 1: Component lifecycle callbacks' execution follows prescribed orders
  - ELITE encodes the ordering as **temporal constraints** and enforce them during method call sequence generation

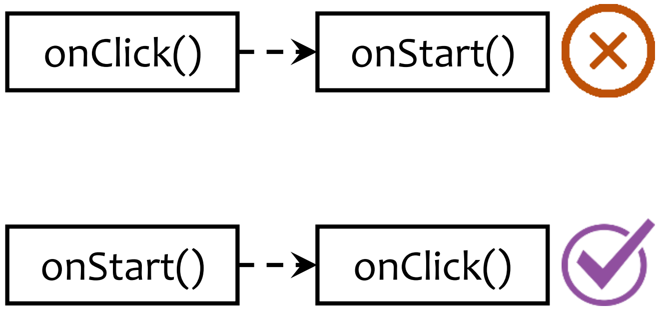


Challenge: Generating Valid Method Call Sequences

- <u>Example 2:</u> GUI and system event callbacks can only be invoked when the corresponding event listeners are registered
  - ELITE **infers each event listener's registering and unregistering methods** (via static analysis) and enforce the proper order during sequence generation

Dynamic registration

```
public class MyActivity extends Activity {
  protected void onStart() {
    Button button = (Button) findViewById(R.id.button_id);
    button.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        // Perform action on click
      }
});}}
```



Summarizing Top Level Methods

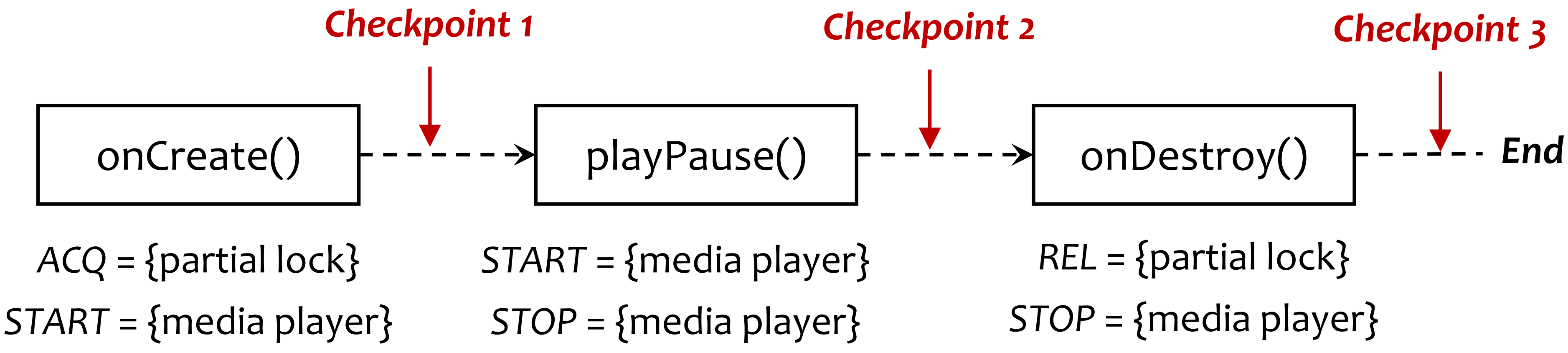
- ELITE encodes potential runtime behaviors of each top level method ***tm*** by means of **four sets of dataflow facts** inferred via **forward inter-procedural dataflow analysis** (performed before sequence generation)

  1. ***ACQ:*** The **wake lock instances** that may have been **acquired** after executing ***tm***
  2. ***REL:*** The **wake lock instances** that may have been **released** after executing ***tm***
  3. ***START:*** The **asynchronous computational tasks** that may have been **started** after executing ***tm***
  4. ***STOP:*** The **asynchronous computational tasks** that may have been **stopped/paused** after executing ***tm***

CASTLE

Wake Lock Necessity Analysis

A top level method call sequence for the app TomaHawk:



Checkpoint: the app may stay quiescent at such state-transitioning time point indefinitely for a long time when there are no events to handle

Wake Lock Necessity Analysis (A Running Example)

- **Intuition:** at each checkpoint, wake locks should be held only if the app is performing long running (asynchronous) computation

Checkpoint

onCreate() - - - - - → playPause() - - - - - → onDestroy() - - - - - ***End***

ACQ = {partial lock}
START = {media player}

START = {media player}
STOP = {media player}

REL = {partial lock}
STOP = {media player}

- **Check 1:** Is it possible after executing playPause() the acquired wake lock would be released? **No**
- **Check 2:** Is it possible after executing playPause(), all started asynchronous computational tasks would be stopped? **Yes** **Since no useful tasks are running, why the wake lock is held?**

Experimental Setup

- Subjects: 12 versions of five large-scale and popular open-source Android apps
  - Six versions contain real wake lock misuses and the other six versions are corresponding issue-fixing versions
- Techniques under comparison
  - **Relda:** a resource leak detection technique for Android apps (Guo et al. ASE'13)
  - **Verifier:** a verification technique for detecting no-sleep bugs (Vekris et al. HotPower'12)

Experimental Result

Our tool

App	Bug type	Version	#Wakelock issues	Result: #TP / #reported warnings		
				ELITE	Relda	Verifier
TomaHawk	Unnecessary wakeup	Buggy	1	1/1	0/1	0/2
		Clean	0	0/0	0/0	0/0
Open-GPSTracker	Unnecessary wakeup	Buggy	1	1/1	0/0	0/1
		Clean	0	0/0	0/0	0/1
MyTracks	Unnecessary wakeup	Buggy	2	2/2	0/1	0/2
		Clean	0	0/0	0/1	0/0
FBReader	Leakage	Buggy	1	1/1	0/0	0/0
		Clean	0	0/0	0/0	0/0
MyTracks	Leakage	Buggy	1	1/1	0/1	0/0
		Clean	0	0/0	0/1	0/0
CallMeter	Leakage	Buggy	1	0/0	0/0	1/1
		Clean	0	0/0	0/0	0/1
Precision				100%	0%	12.5%
Recall				85.7%	0%	14.3%

False alarm

Experimental Result Analysis

- Relda and Verifier: high rate of false positives/negatives
  - Rely on pre-defined wake lock releasing points (e.g., Activity.onPause() handler) for leakage detection, oblivious to app semantics and runtime behaviors

Wake lock releasing points in activities (results of analyzing 44,736 apps)

Releasing point	Percentage of apps
onPause()	**35.4%**
onDestroy()	15.8%
onResume()	13.0%
onWindowFocusChanged()	11.2%
onCreate()	10.2%
Other 389 callbacks	14.4%

Experimental Result Analysis

- Relda and Verifier: high rate of false positives/negatives
  - Rely on pre-defined wake lock releasing points (e.g., Activity.onPause() handler) for leakage detection, oblivious to app semantics and runtime behaviors
  - Do not locate all defined program callbacks and properly handle the execution order among them (ELITE systematically locates all callbacks with a fix-point iterative algorithm and infer temporal constraints to model callback execution orders)
  - Lack of full path sensitivity in program analysis (also the reason for ELITE's false negative when analyzing CallMeter)

Conclusion

- The first large-scale empirical study of wake lock usage in practice
- Eight common patterns of wake lock misuses
- ELITE: a static analysis technique for wake lock misuse detection
- A preliminary evaluation shows that ELITE outperforms existing techniques

Our datasets and tool are available at:

http://sccpu2.cse.ust.hk/elite/

Artifact Evaluated