Finding Sensor Related Energy Black Holes In Smartphone Applications

Yepang Liu[1], Chang Xu[2], S.C. Cheung[1]

[1]The Hong Kong University of Science and Technology

[2]State Key Lab for Novel Software, Nanjing University

IEEE PerCom 2013

San Diego, California, USA

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

南京大學
NANJING UNIVERSITY

Smartphone apps

Google play

650,000+ applications
(September 2012)

25+ billion downloads
(September 2012)

Energy Problem

Full Network access

Frequent sensor usage

3D rendering

Energy Problem

Full Network access

Frequent sensor usage

3D rendering

Annual density improvement very slow (6%)

Energy Inefficiency

Energy Problem

- Problem magnitude
  - Thousands of apps are NOT energy efficient
  - Millions of users affected and complained
  - Phone batteries drained in a few hours

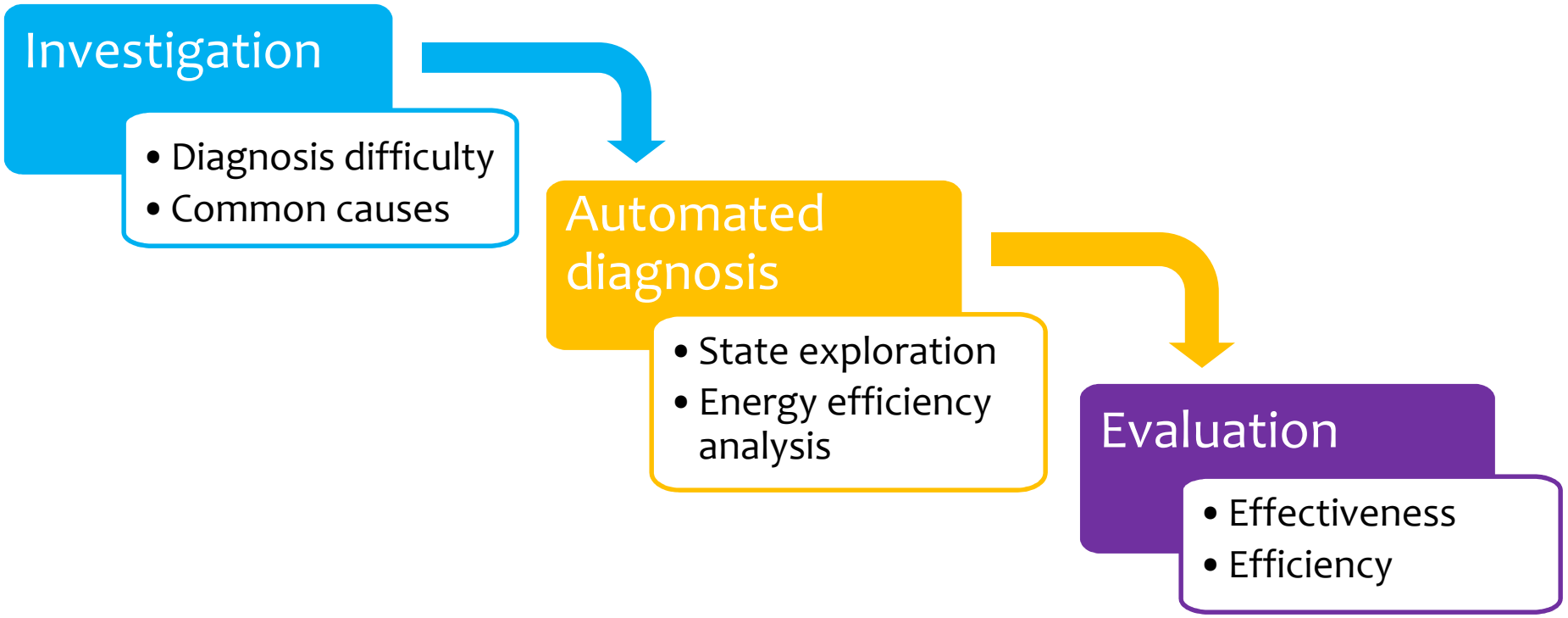

(Pathak et al. Hotnets 2011)

- Major reasons
  - Hardware management burden (e.g., sensors)
  - Lack of dedicated QA, short time to market
  - Difficulty in problem diagnosis

Motivation

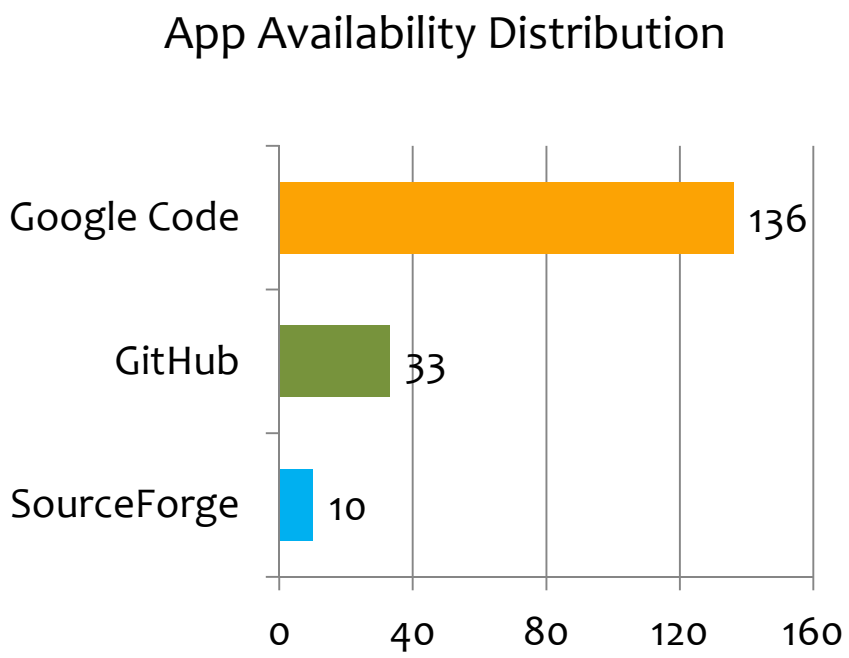
- What are the common causes of energy problems?
- Can we distill patterns to enable automated diagnosis?

Our Work


Investigation
• Diagnosis difficulty
• Common causes
Automated diagnosis
• State exploration
• Energy efficiency analysis
Evaluation
• Effectiveness
• Efficiency

Investigated Subjects

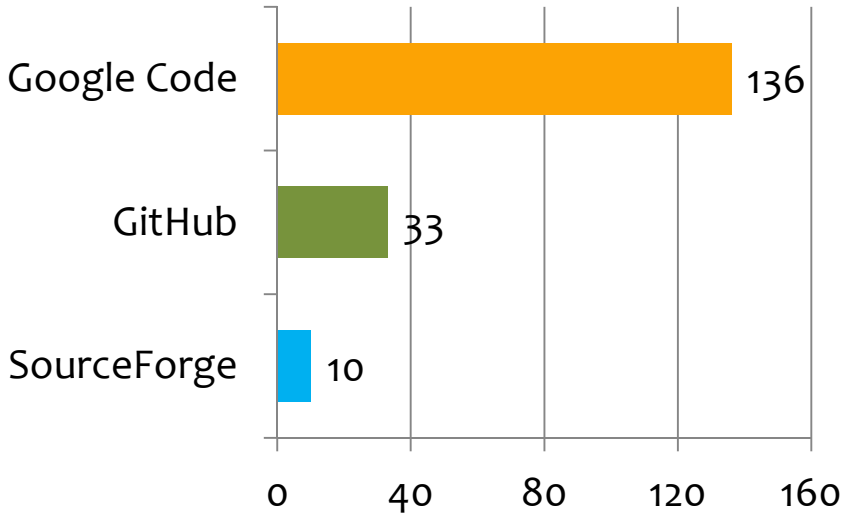
174 popular open-source Android apps



Investigated Subjects

174 popular open-source Android apps

App Availability Distribution



Source Repository (24 affected apps)
✓ Bug reports
✓ Comments on bug reports
✓ Bug fixing patches
✓ Revision commit logs

Observations

Diagnosis difficulty

- Reproduce problem (extensive testing, energy profiling)
- Figure out root cause (instrumentation, runtime logging)

Observations

Diagnosis difficulty

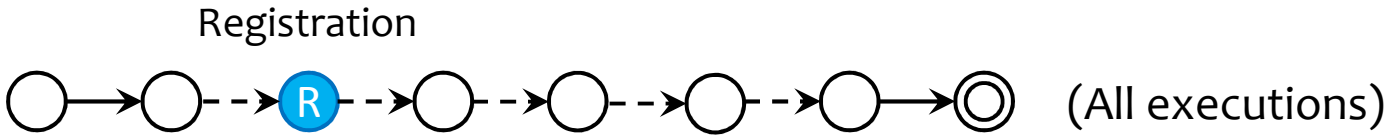
- Reproduce problem (extensive testing, energy profiling)
- Figure out root cause (instrumentation, runtime logging)

Problem causes

- Common causes (10/24): improper use of sensors

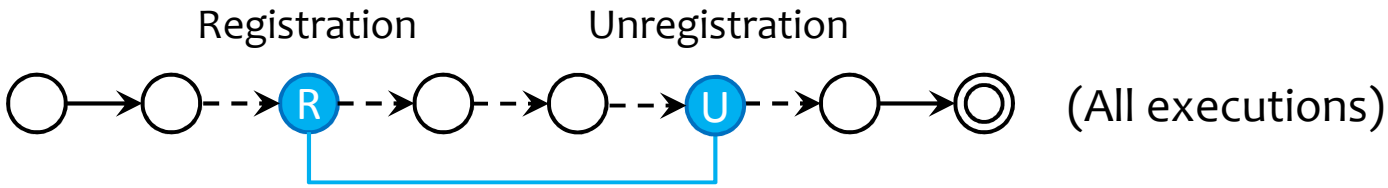
Patterns

- Sensor listener misusage



Patterns

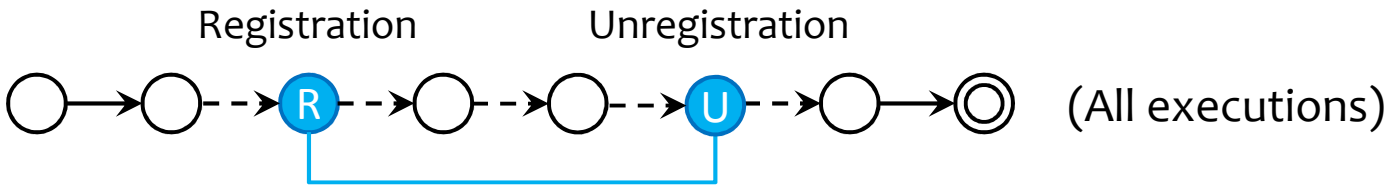
- Sensor listener misusage



"Always un-register sensor listener before program exits!"

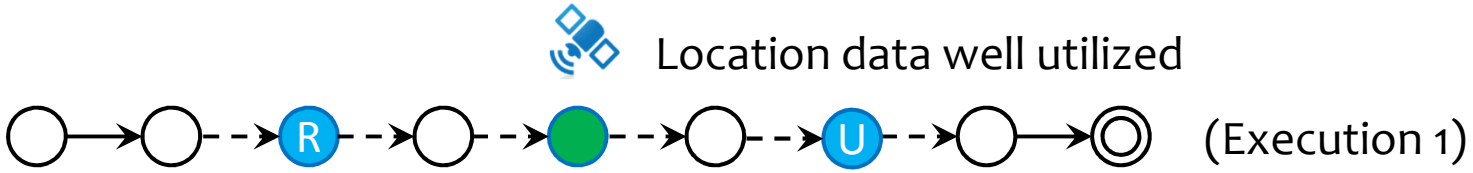
Patterns

- Sensor listener misusage



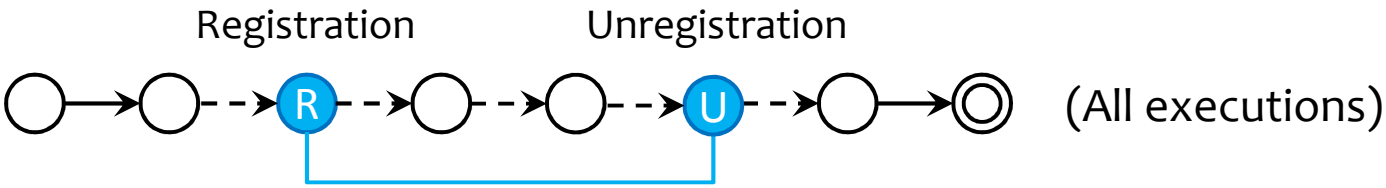
"Always un-register sensor listener before program exits!"

- Sensory data underutilization



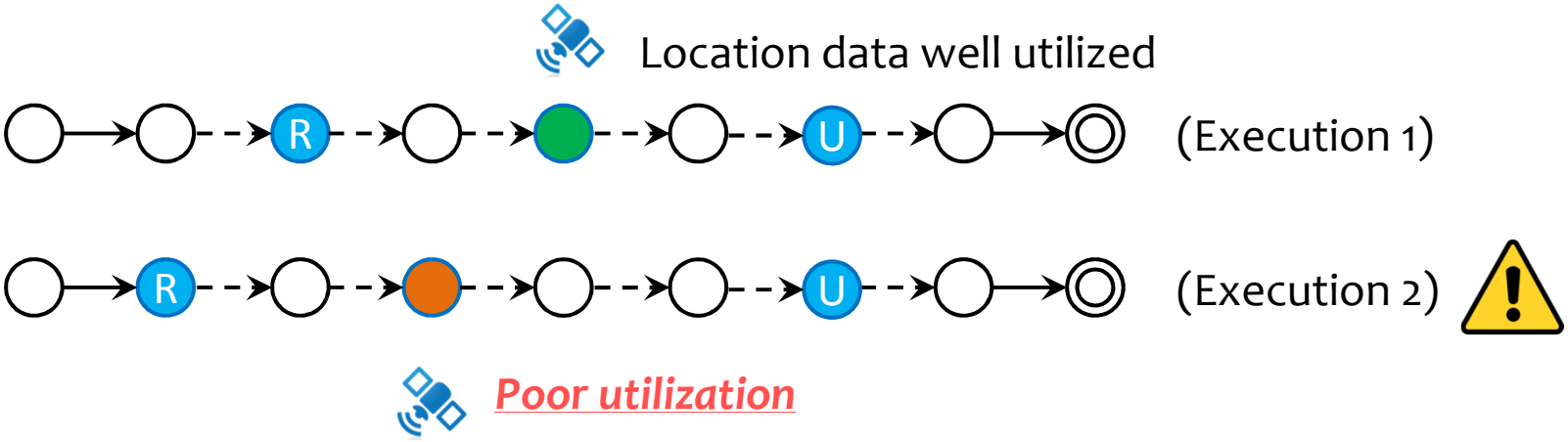
Patterns

- Sensor listener misusage

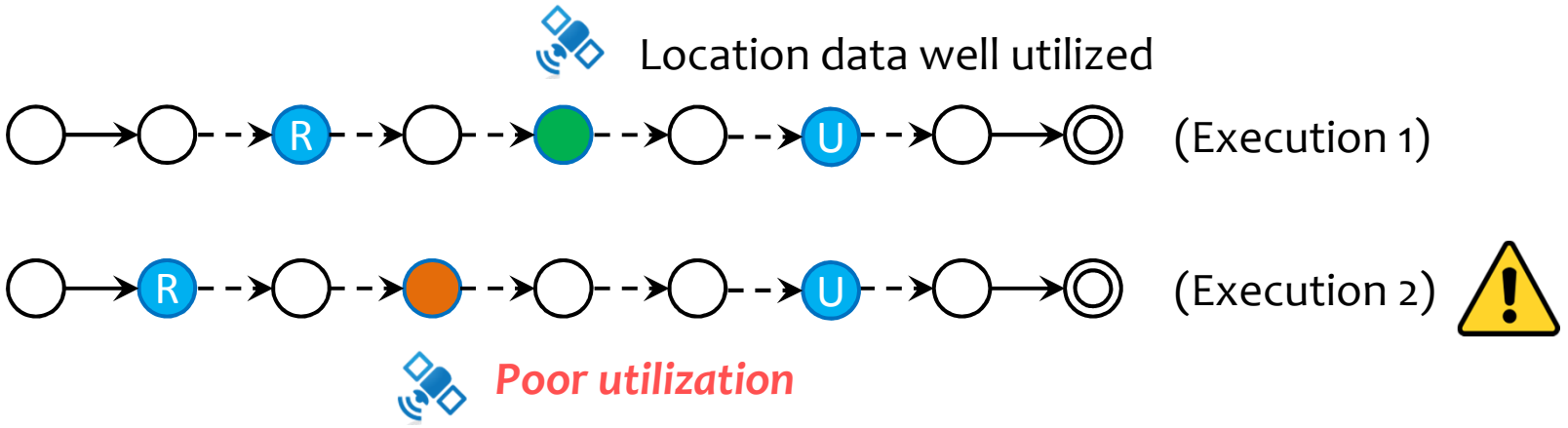


"Always un-register sensor listener before program exits!"

- Sensory data underutilization



Sensory Data Underutilization



"GeoHashDroid should slow down sensor update significantly if nothing besides the notification bar is listening."

(GeoHashDroid Issue 24)

"GPS sensor should be timely disabled if location data are used to update an invisible map."

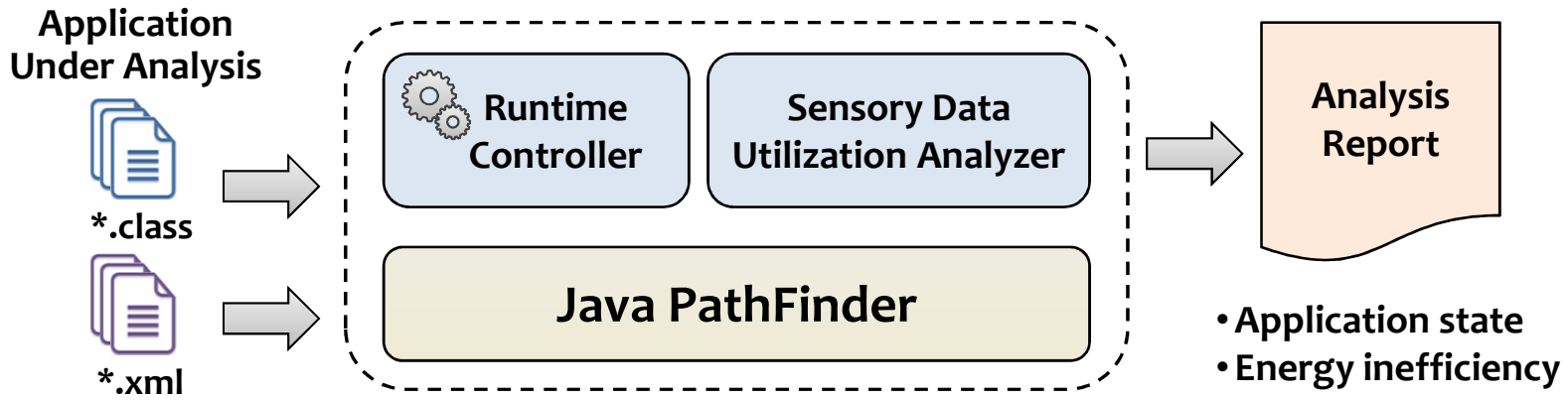
(Osmdroid Issue 53)

Approach Overview (GreenDroid)

- Dynamic analysis (on top of Java PathFinder)
- Goal: Simple, scalable, and effective

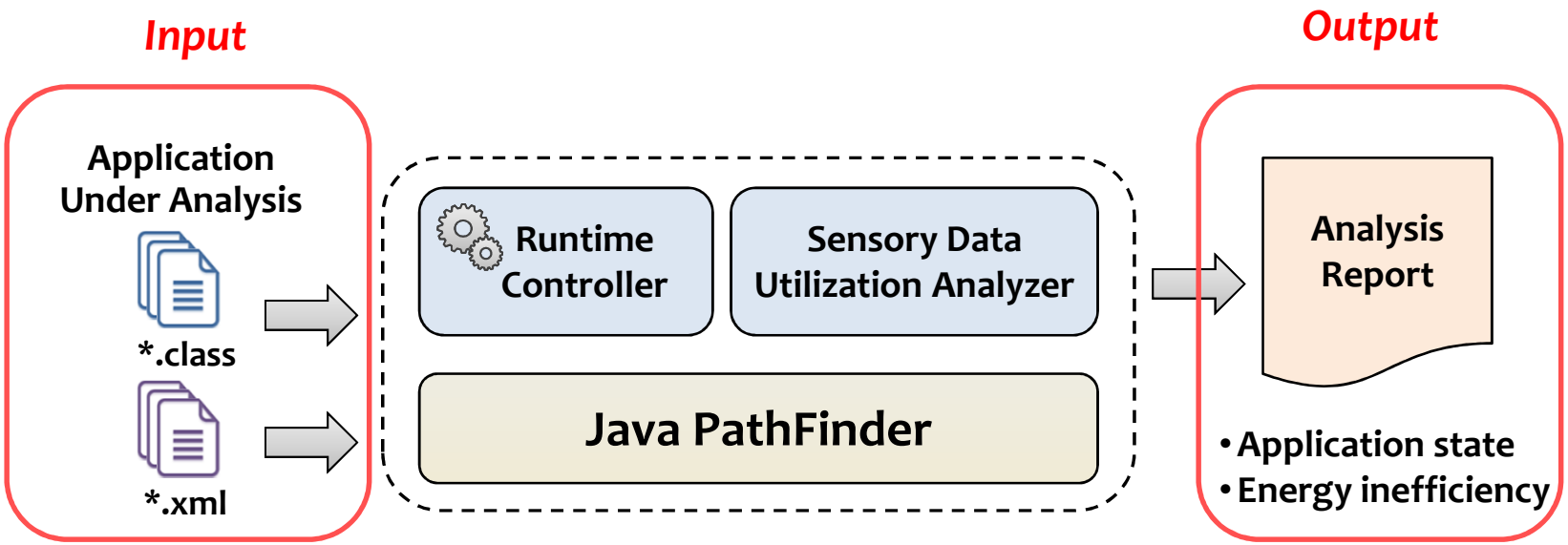
Approach Overview (GreenDroid)

- Dynamic analysis (on top of Java PathFinder)
- Goal: Simple, scalable, and effective



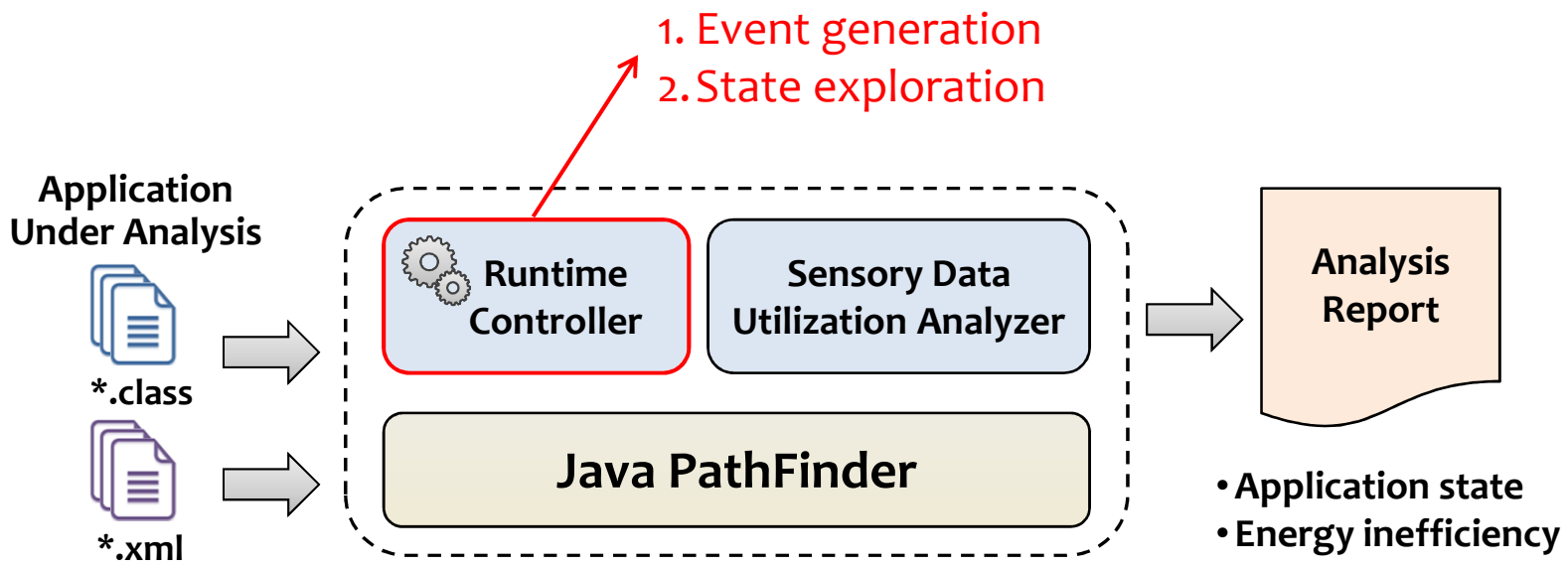
Approach Overview (GreenDroid)

- Dynamic analysis (on top of Java PathFinder)
- Goal: Simple, scalable, and effective



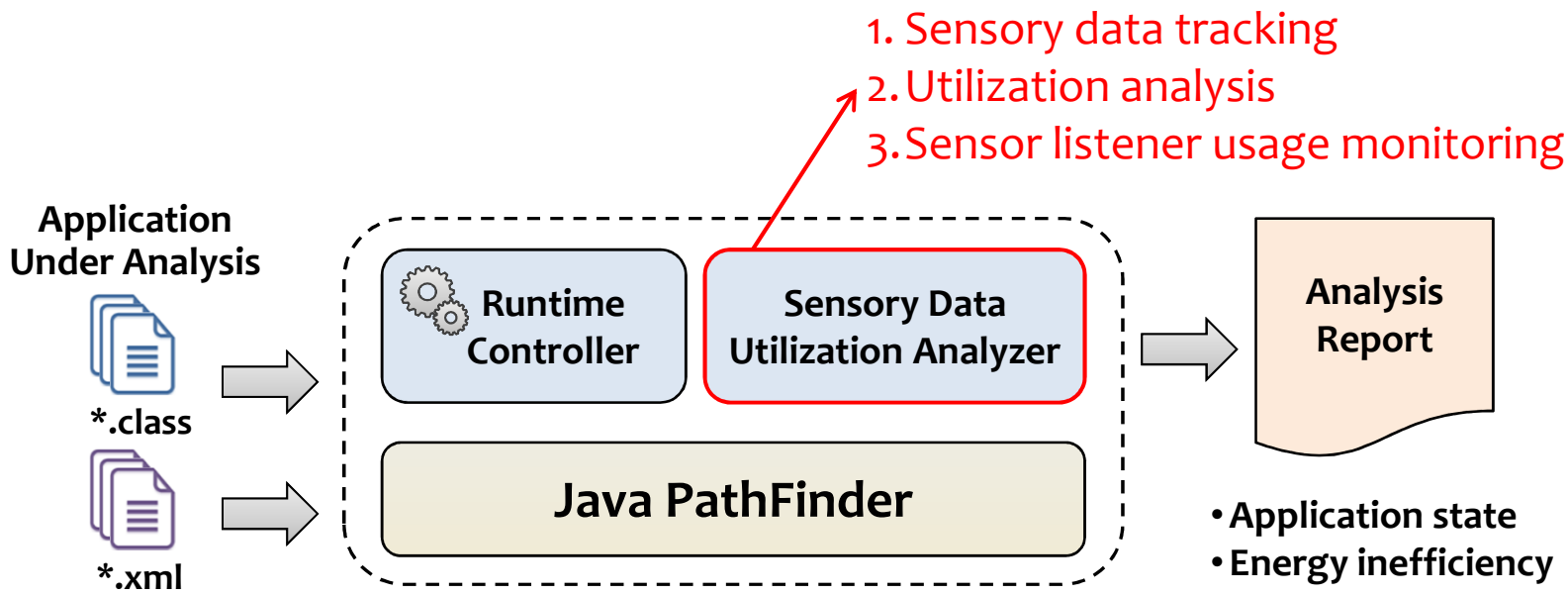
Approach Overview (GreenDroid)

- Dynamic analysis (on top of Java PathFinder)
- Goal: Simple, scalable, and effective



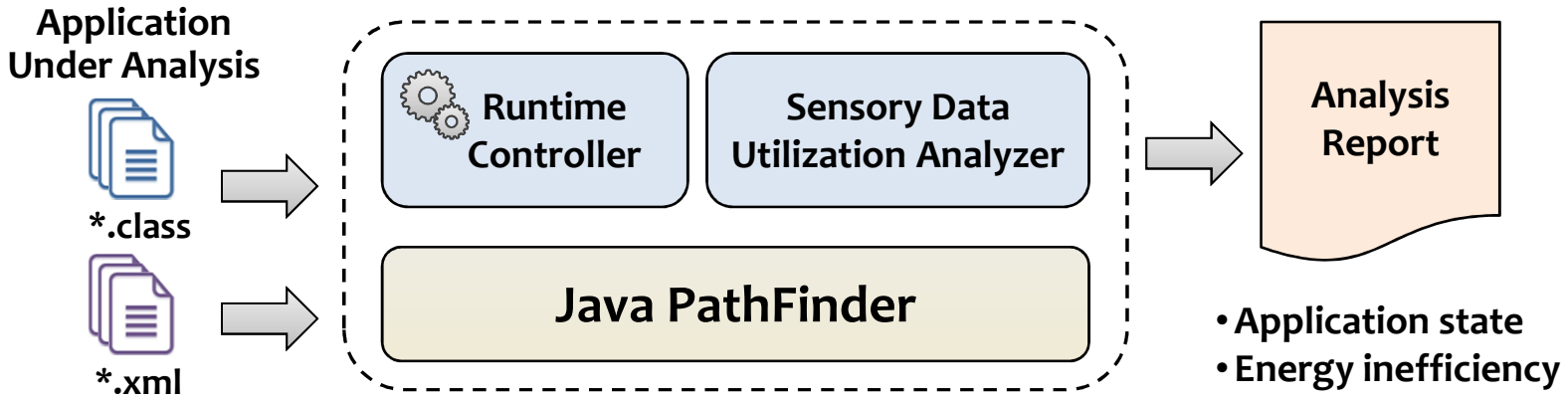
Approach Overview (GreenDroid)

- Dynamic analysis (on top of Java PathFinder)
- Goal: Simple, scalable, and effective



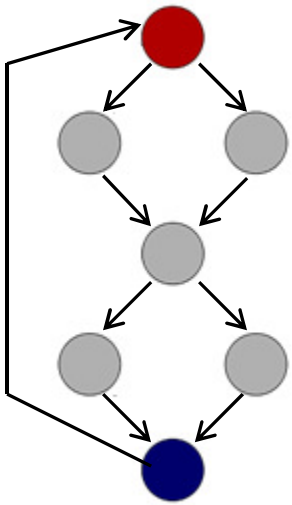
Approach Overview (GreenDroid)

- Dynamic analysis (on top of Java PathFinder)
- Goal: Simple, scalable, and effective
- Major Challenges
  - App execution and state exploration in Java PathFinder
  - Sensory data identification and utilization analysis (no metrics)



App Execution in JPF (Problems)

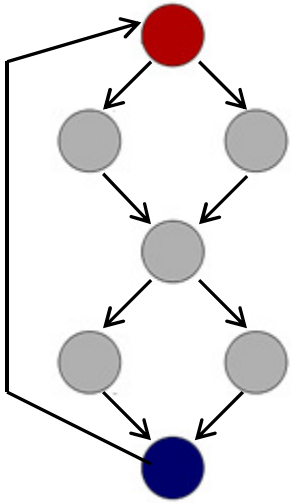
- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



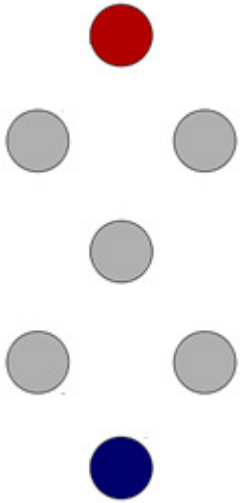
General Java programs
(explicit control flow)

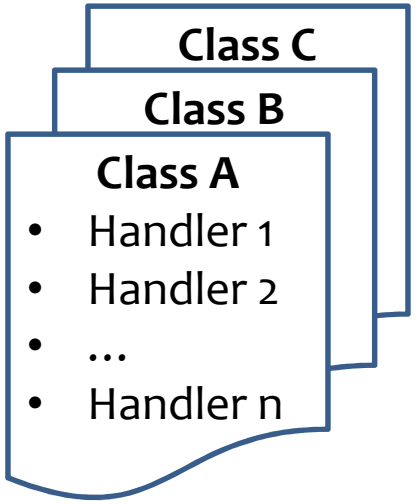
App Execution in JPF (Problems)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



VS





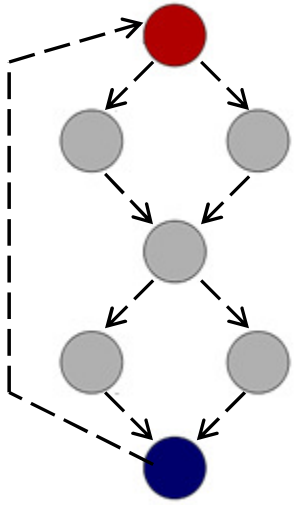
General Java programs
(explicit control flow)

Android programs
(event-driven)

Loosely coupled handlers

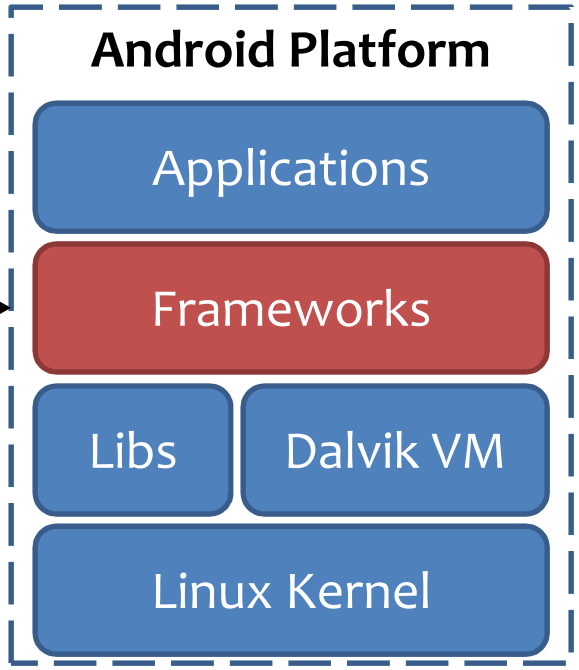
App Execution in JPF (Problems)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



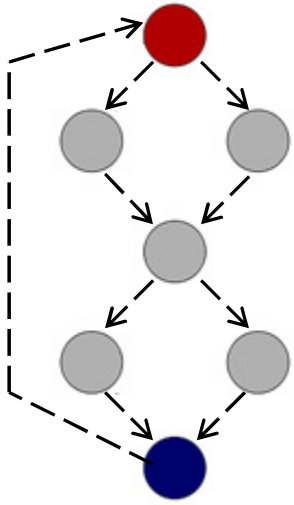
Implicit control flow

Scheduled by



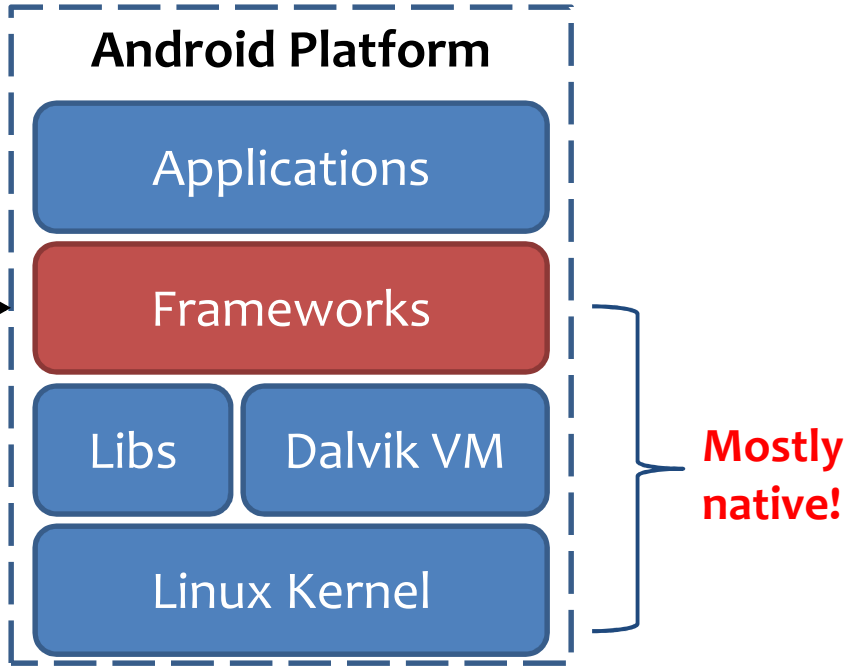
App Execution in JPF (Problems)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



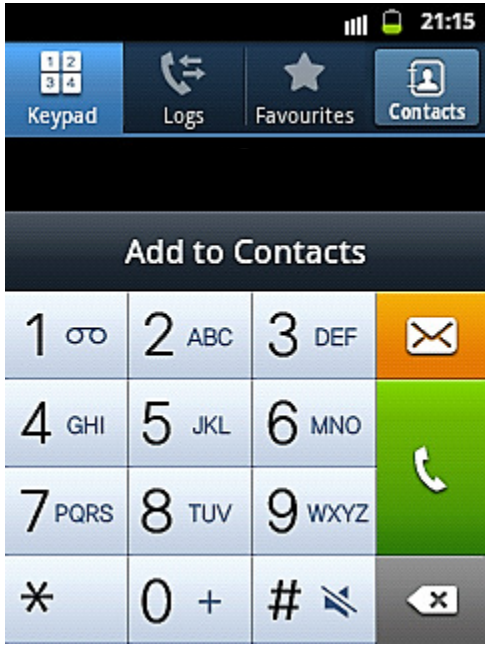
Implicit control flow

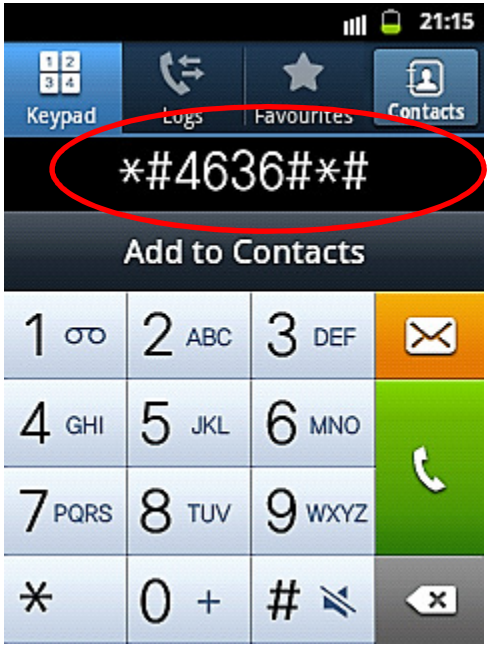
Scheduled by



App Execution in JPF (Problems)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



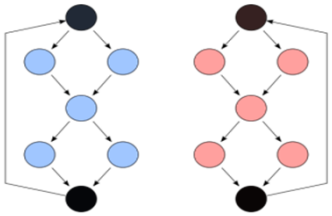


App Execution in JPF (Solutions)

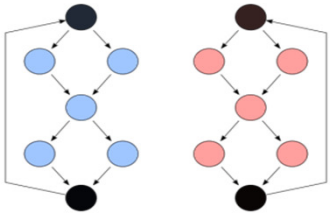
- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)

Android Specs

Handler scheduling policies



Temporal rules (AEM Model)



Temporal rules (AEM Model)

Concretization

Input: (1) app execution history
(2) Newly received event

Output: next handler to execute

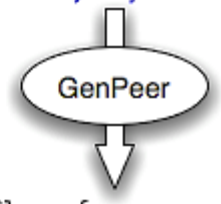
Decision procedure

App Execution in JPF (Solutions)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)

```
package x.y.z;
class MyClass {
  ...
  native String foo (int i, String s);
}
```

"java gov.nasa.jpf.GenPeer x.y.z.MyClass > JPF_x_y_z_MyClass.java"




```
class JPF_x_y_z_MyClass {
  ...
  public static
      int foo__ILjava_lang_String__2 (MJIEnv env, int objRef,
                                      int i, int sRef) {
    int ref = MJIEnv.NULL;
    // <2do> fill in body
    return ref;
  }
}
```

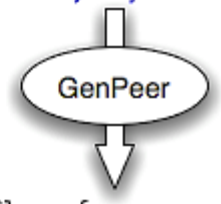
App Execution in JPF (Solutions)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)

Native method

```
package x.y.z;
class MyClass {
  ...
  native String foo (int i, String s);
}
```

"java gov.nasa.jpf.GenPeer x.y.z.MyClass > JPF_x_y_z_MyClass.java"




```
class JPF_x_y_z_MyClass {
  ...
  public static
      int foo__ILjava_lang_String__2 (MJIEnv env, int objRef,
                                      int i, int sRef) {
    int ref = MJIEnv.NULL;
    // <2do> fill in body
    return ref;
  }
}
```

Identify native methods

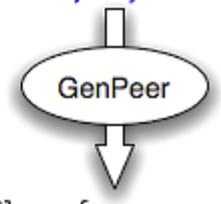
App Execution in JPF (Solutions)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)

Native method

```
package x.y.z;
class MyClass {
  ...
  native String foo (int i, String s);
}
```

"java gov.nasa.jpf.GenPeer x.y.z.MyClass > JPF_x_y_z_MyClass.java"



Native peer

```
class JPF_x_y_z_MyClass {
  ...
  public static
      int foo__ILjava_lang_String__2 (MJIEnv env, int objRef,
                                      int i, int sRef) {
    int ref = MJIEnv.NULL;
    // <2do> fill in body
    return ref;
  }
}
```

Identify native methods

Create stubs (native peers)

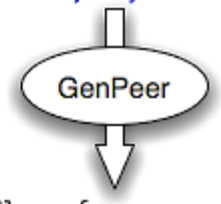
App Execution in JPF (Solutions)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)

Native method

```
package x.y.z;
class MyClass {
  ...
  native String foo (int i, String s);
}
```

"java gov.nasa.jpf.GenPeer x.y.z.MyClass > JPF_x_y_z_MyClass.java"



Native peer

```
class JPF_x_y_z_MyClass {
  ...
  public static
      int foo__ILjava_lang_String__2 (MJIEnv env, int objRef,
                                      int i, int sRef) {
    int ref = MJIEnv.NULL;
    // <2do> fill in body
    return ref;
  }
}
```

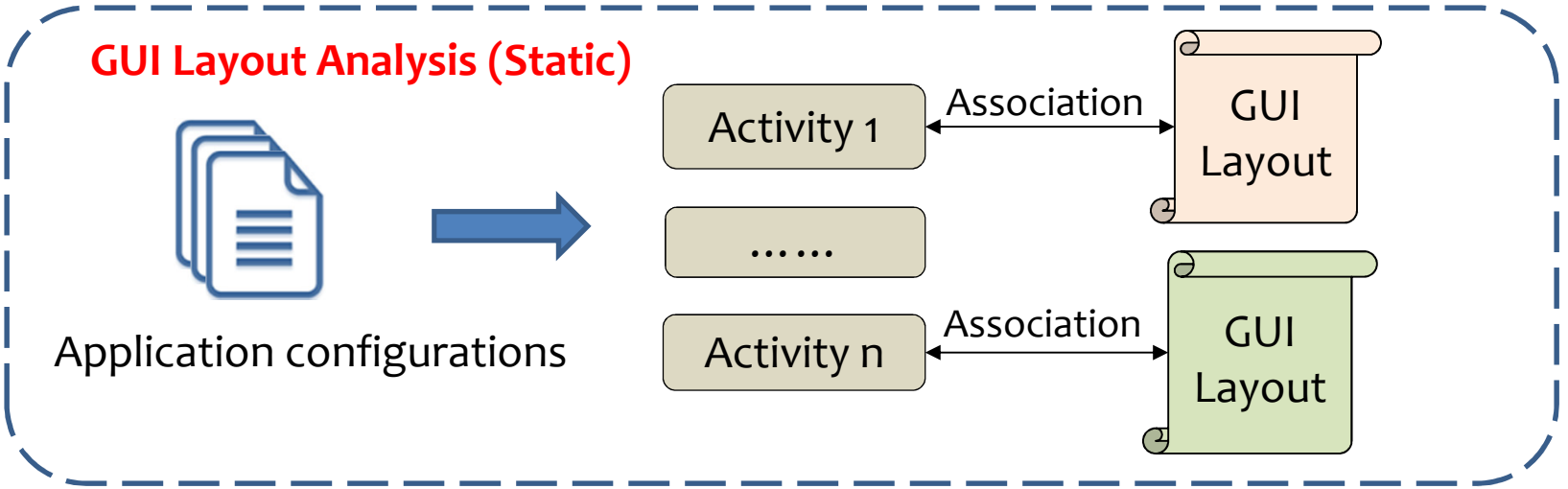
Identify native methods

Create stubs (native peers)

Implement logics

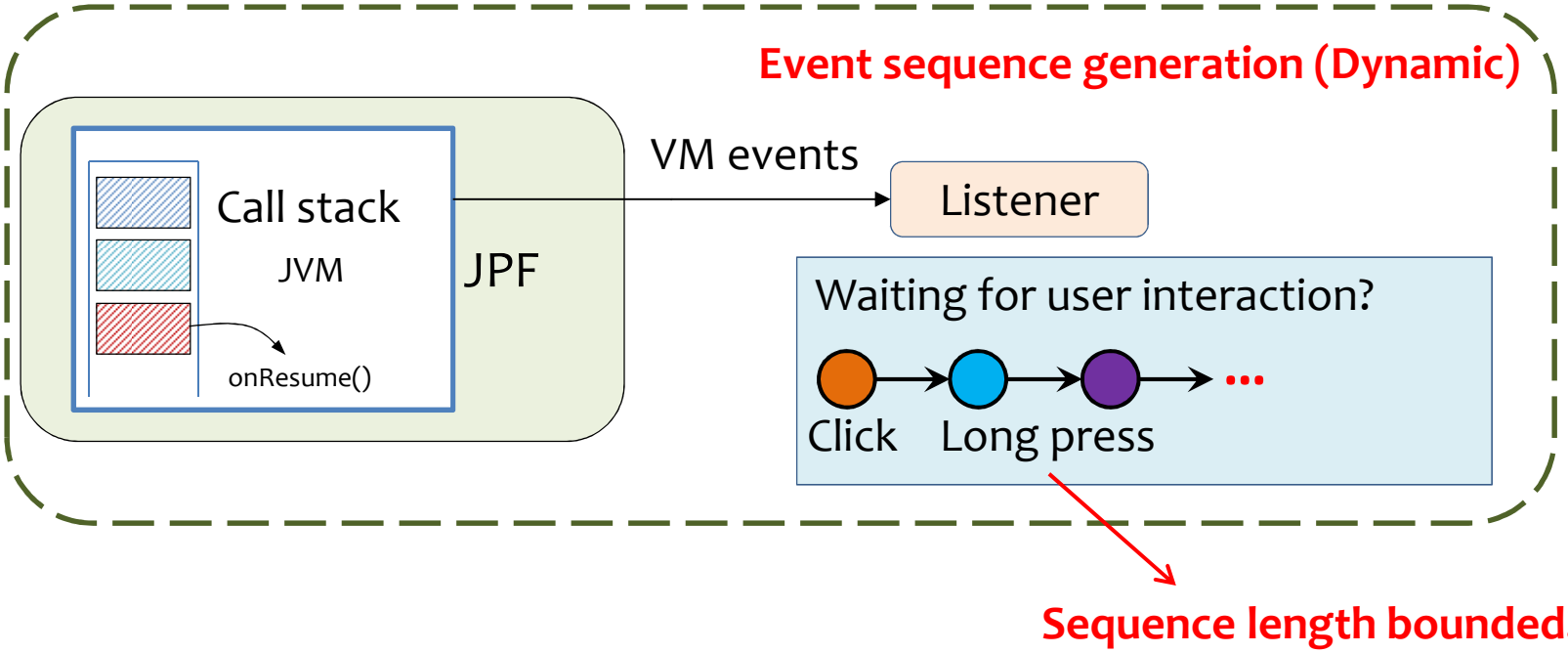
App Execution in JPF (Solutions)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



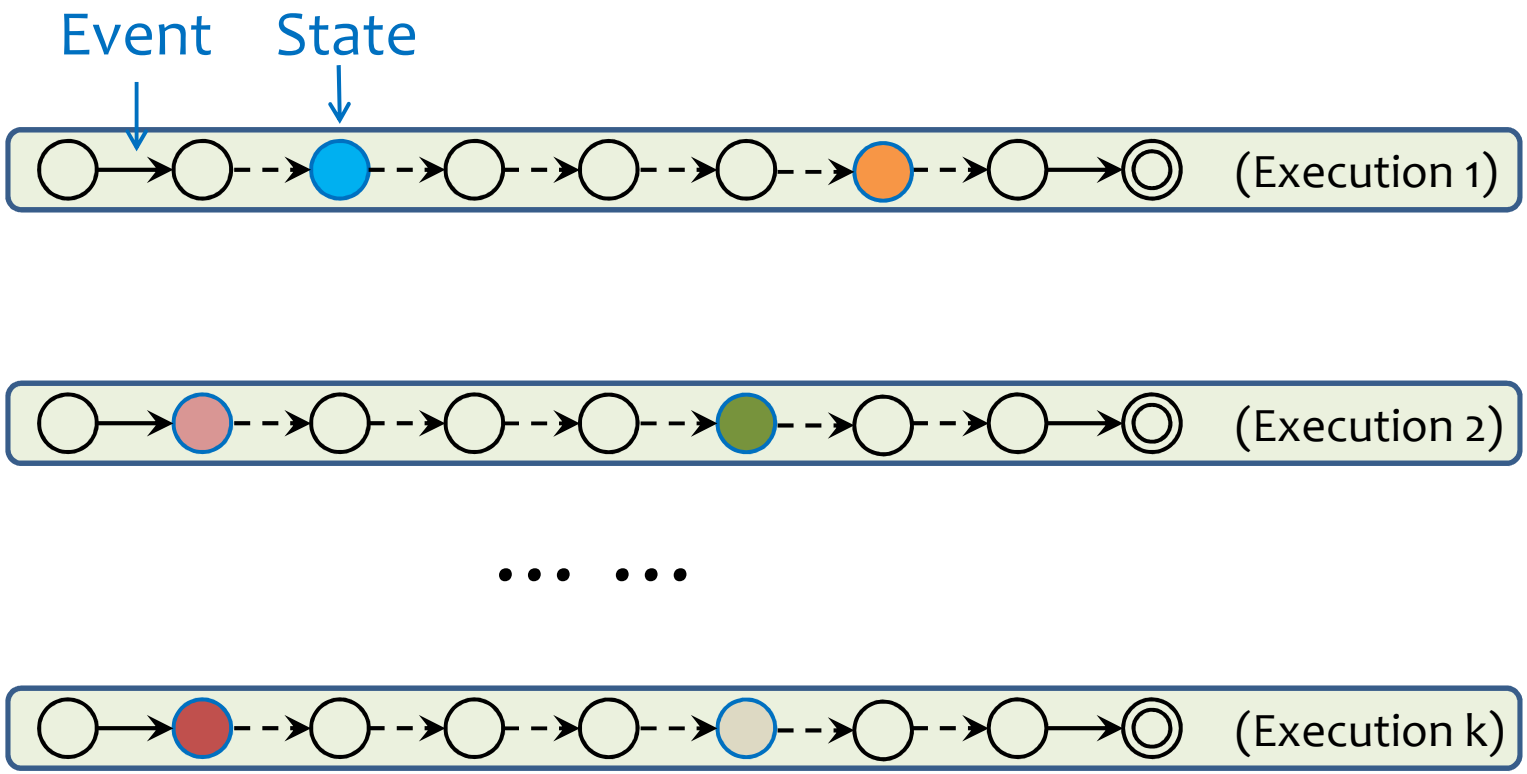
App Execution in JPF (Solutions)

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



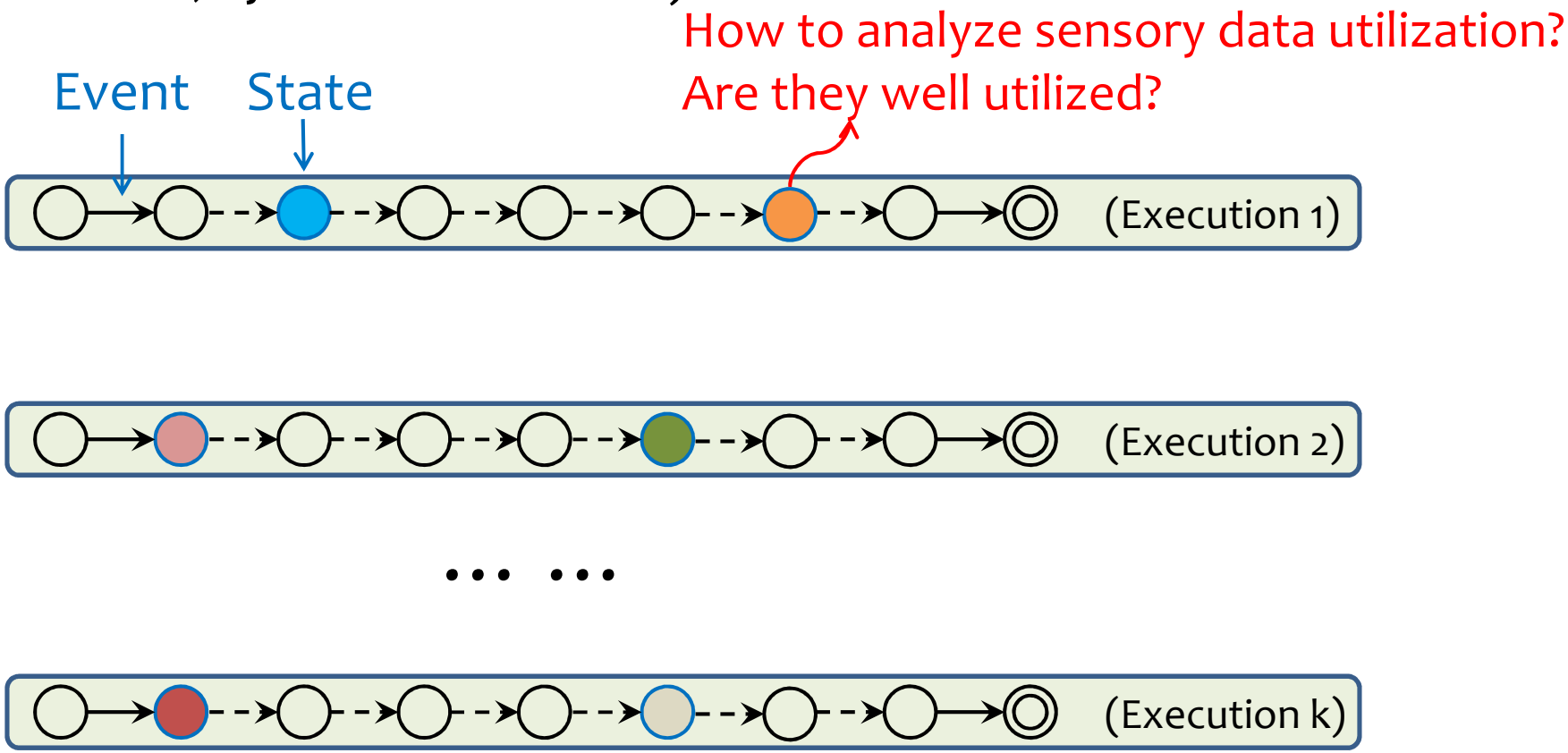
State Exploration

- State changes as the app continuous handles events (user events, system events etc.)



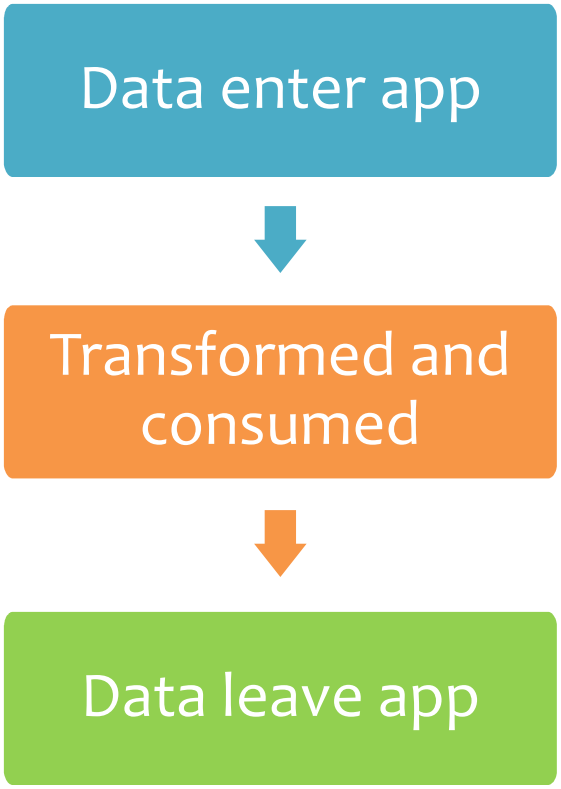
State Exploration

- State changes as the app continuous handles events (user events, system events etc.)



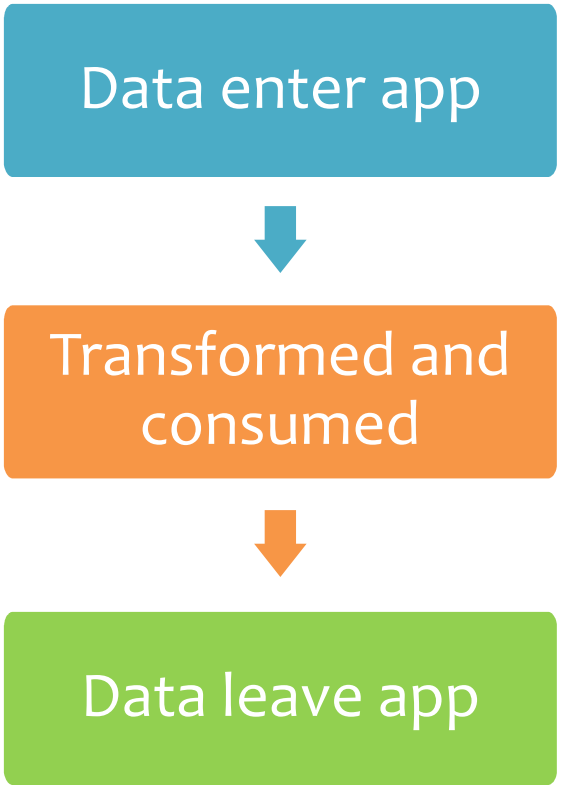
Sensory Data Tracking & Identification

Data Lifecycle:



Sensory Data Tracking & Identification

Data Lifecycle:



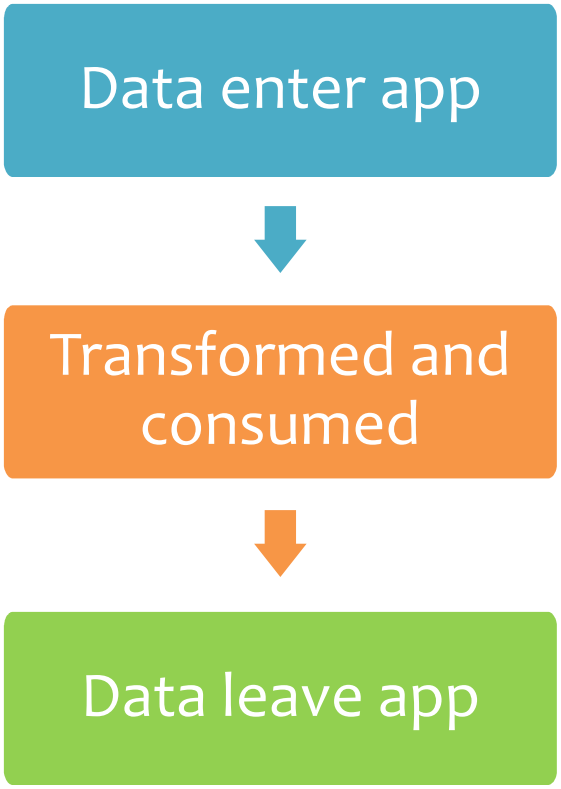
Sensory Data tracking process:

Taint source

- Taint the mock sensory data

Sensory Data Tracking & Identification

Data Lifecycle:



Sensory Data tracking process:

Taint source

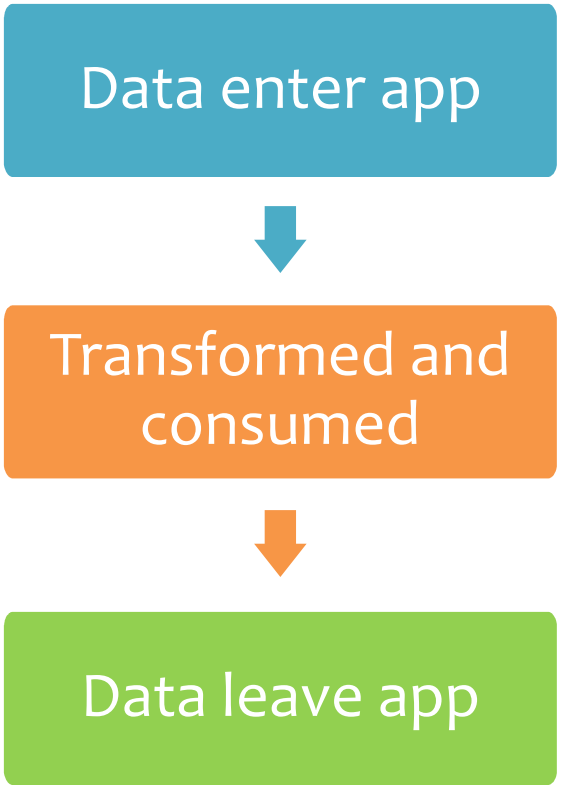
- Taint the mock sensory data

Taint propagation

- Tracking and pinpointing sensory data
- Record sensory data usages

Sensory Data Tracking & Identification

Data Lifecycle:



Sensory Data tracking process:

Taint source

- Taint the mock sensory data

Taint propagation

- Tracking and pinpointing sensory data
- Record sensory data usages

Taint sink

- Terminate tainting
- Compute sensory data utilization

Taint Propagation Policy

Index	Bytecode Instruction	Taint Propagation Rule
1	***Const-op*** *C*	$T(stack[0]) = \emptyset$
2	***Load-op*** *index*	$T(stack[0]) = T(localVar_{index})$
3	***LoadArray-op*** *arrayRef, index*	$T(stack[0]) = T(arrayRef) \cup T(arrayRef\ [index])$
4	***Store-op*** *index*	$T(localVar_{index}) = T(stack'[0])$
5	***StoreArray-op*** *arrayRef, index*	$T(arrayRef\ [index]) = T(stack'[0])$
6	***Binary-op***	$T(stack[0]) = T(stack'[0]) \cup T(stack'[1])$
7	***Unary-op***	$T(stack[0]) = T(stack'[0])$
8	***GetField-op*** *index*	$T(stack[0]) = T(stack'[0].instanceField) \cup T(stack'[0])$
9	***GetStatic-op*** *index*	$T(stack[0]) = T(ClassName.staticField)$
10	***PutField-op*** *index*	$T(stack'[1].instanceField) = T(stack'[0])$
1	***PutStatic-op*** *index*	$T(ClassName.staticField) = T(stack'[0])$
12	***Return-op(non-void)***	$T(callerStack[0]) = T(calleeStack'[0])$

Example

Compute acceleration

Input: accEvent from accelerometer

```
float[] values = accEevent.values;
```

↓

```
float x = values[0];
float y = values[1];
float z = values[2];
```

↓

```
float g = GRAVERTIY_EARTH;
```

↓

```
float acc = (x*x + y*y + z*z) / (g*g);
```

Example

Tainted Data
accEvent

Compute acceleration

Input: accEvent from accelerometer

```
float[] values = accEevent.values;
        ↓
float x = values[0];
float y = values[1];
float z = values[2];
        ↓
float g = GRAVERTIY_EARTH;
        ↓
float acc = (x*x + y*y + z*z) / (g*g);
```

Example

Tainted Data
accEvent
values (Rule 8)

Compute acceleration

Input: accEvent from accelerometer

Field access

```
float[] values = accEevent.values;
```

↓

```
float x = values[0];
float y = values[1];
float z = values[2];
```

↓

```
float g = GRAVERTIY_EARTH;
```

↓

```
float acc = (x*x + y*y + z*z) / (g*g);
```

Example

Tainted Data
accEvent
values (Rule 8)
x (Rule 3)
y (Rule 3)
z (Rule 3)

Compute acceleration

Input: accEvent from accelerometer

```
float[] values = accEevent.values;
        ↓
float x = values[0];
float y = values[1];
float z = values[2];
        ↓
float g = GRAVERTIY_EARTH;
        ↓
float acc = (x*x + y*y + z*z) / (g*g);
```

Array element access

Example

Tainted Data
accEvent
values (Rule 8)
x (Rule 3)
y (Rule 3)
z (Rule 3)
acc (Rule 6)

Compute acceleration

Input: accEvent from accelerometer

```
float[] values = accEevent.values;
```

↓

```
float x = values[0];
float y = values[1];
float z = values[2];
```

↓

```
float g = GRAVERTIY_EARTH;
```

↓

```
float acc = (x*x + y*y + z*z) / (g*g);
```

Arithmetic computation

Sensory data usage measurement

$$usage(s,d) = \sum_{i \in Instr(s,d)} weight(i,s) \times rel(i)$$

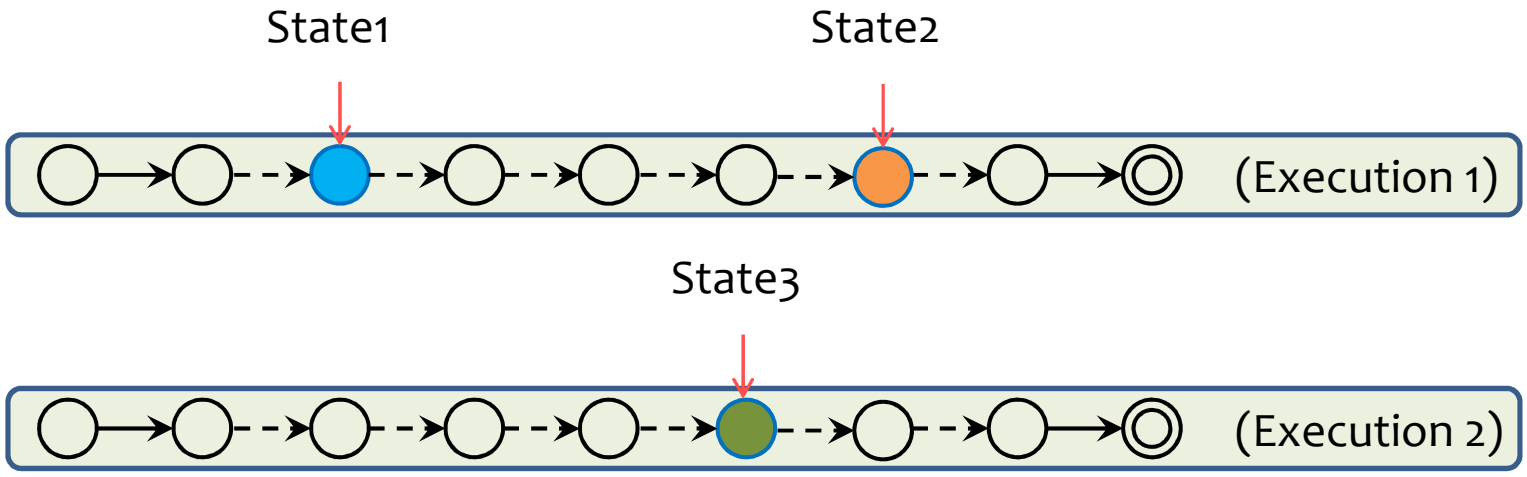
Usage Accumulation

Sensory data usage measurement

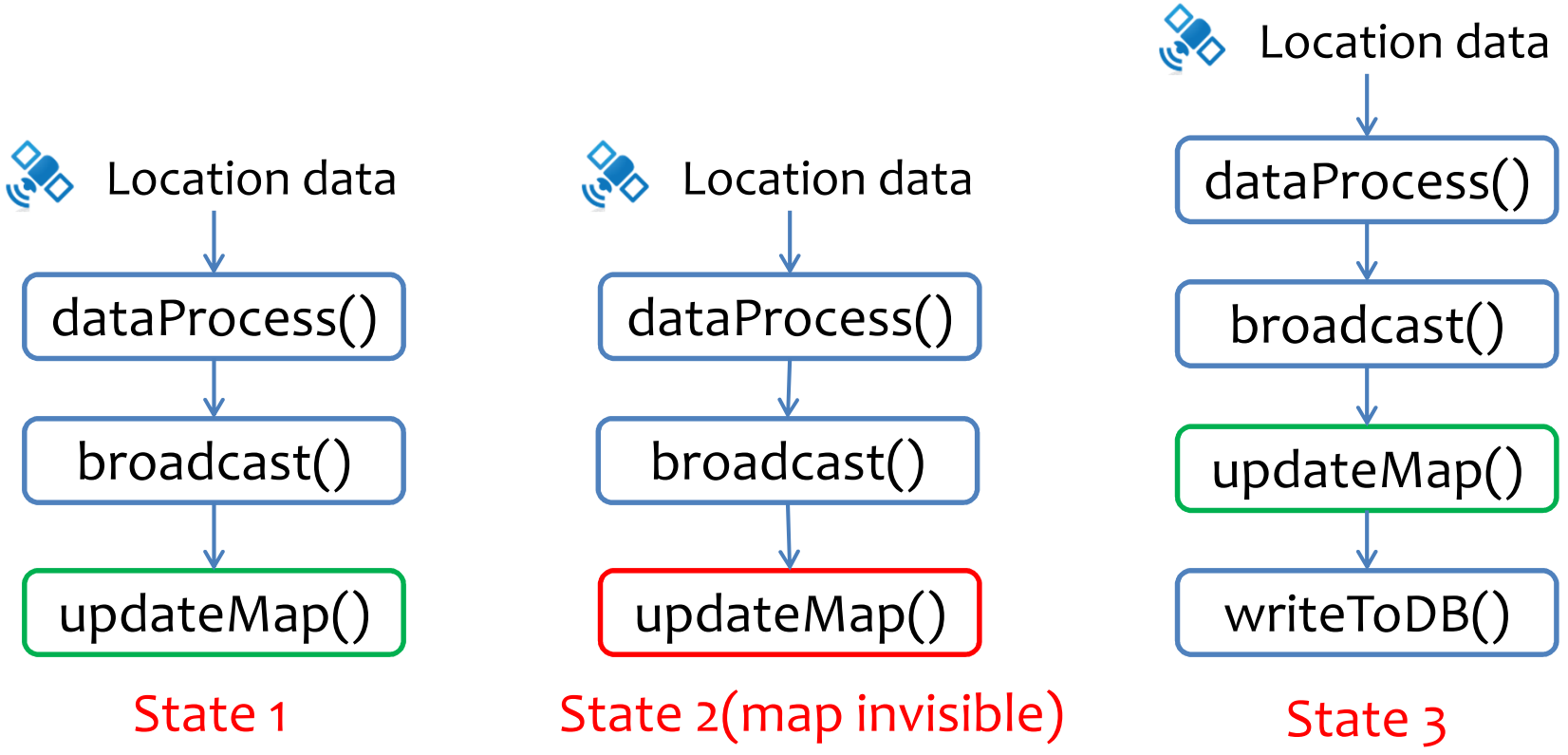
$$usage(s,d) = \sum_{i \in Instr(s,d)} weight(i,s) \times rel(i)$$

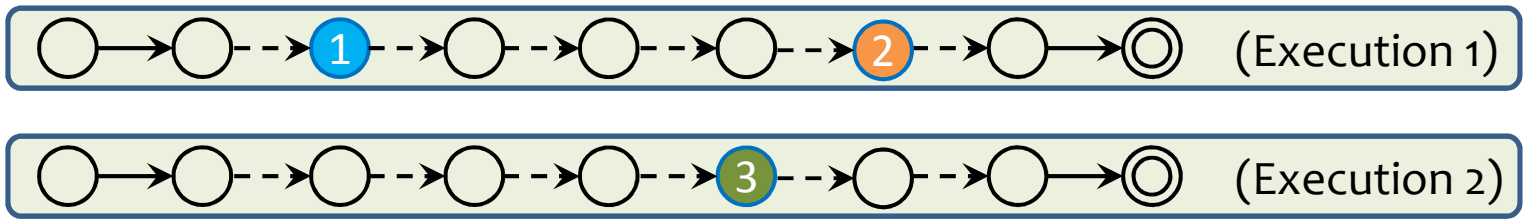
Usage Accumulation

Osmdroid issue 53:

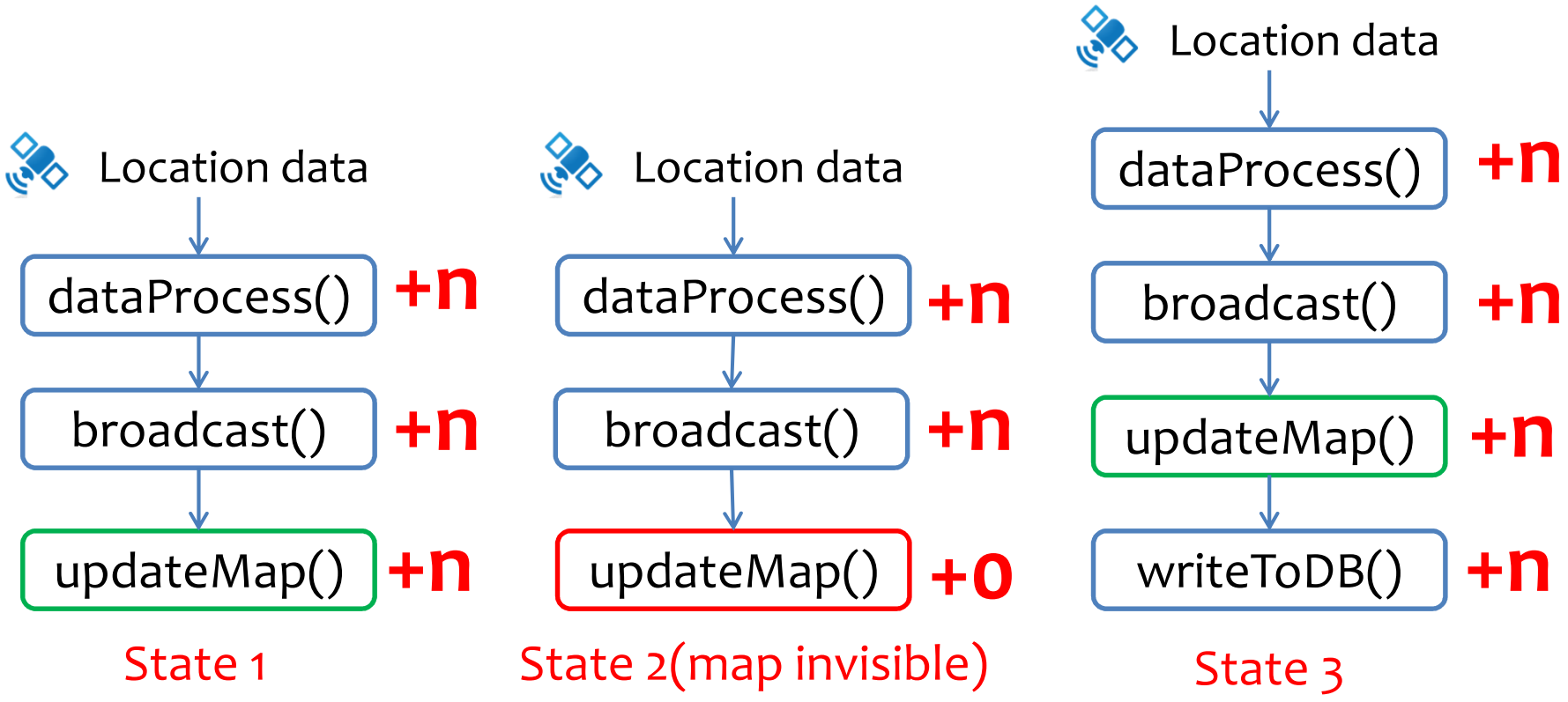


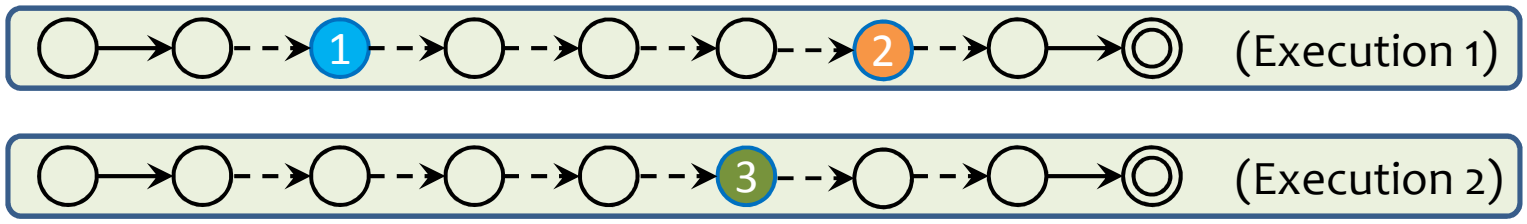
Sensory data usage measurement





Sensory data usage measurement

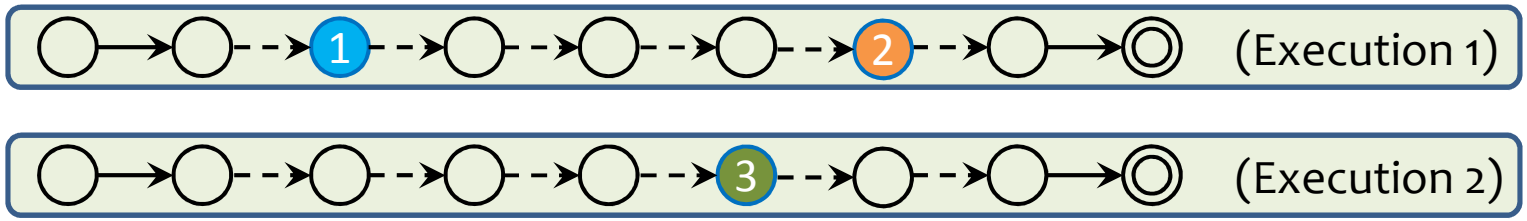




Usage Comparison

$$utilization\_coefficient(s,d) = \frac{usage(s,d)}{Max_{s' \in S, d' \in D}(usage(s',d'))}$$

Index	Usage	Utilization coefficient
State 1	3n	0.75
State 2	**2n**	**0.5**
State 3	4n	1



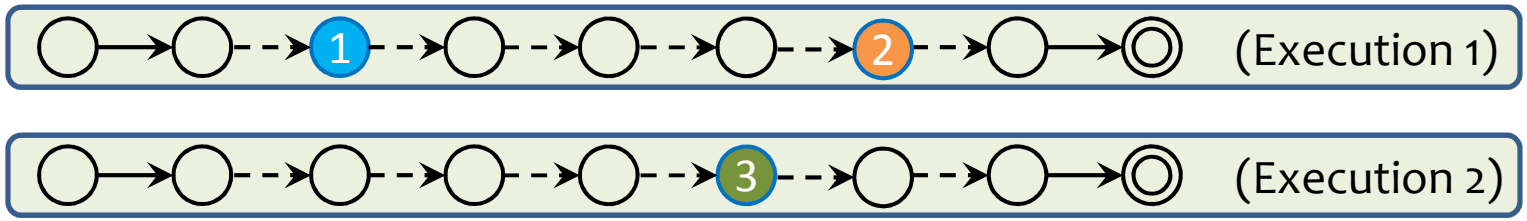
Usage Comparison

$$utilization\_coefficient(s,d) = \frac{usage(s,d)}{Max_{s' \in S, d' \in D}(usage(s',d'))}$$

Index	Usage	Utilization coefficient
State 1	3n	0.75
State 2	**2n**	**0.5**
State 3	4n	1

Report

- Event sequence
- Sensory data usage details



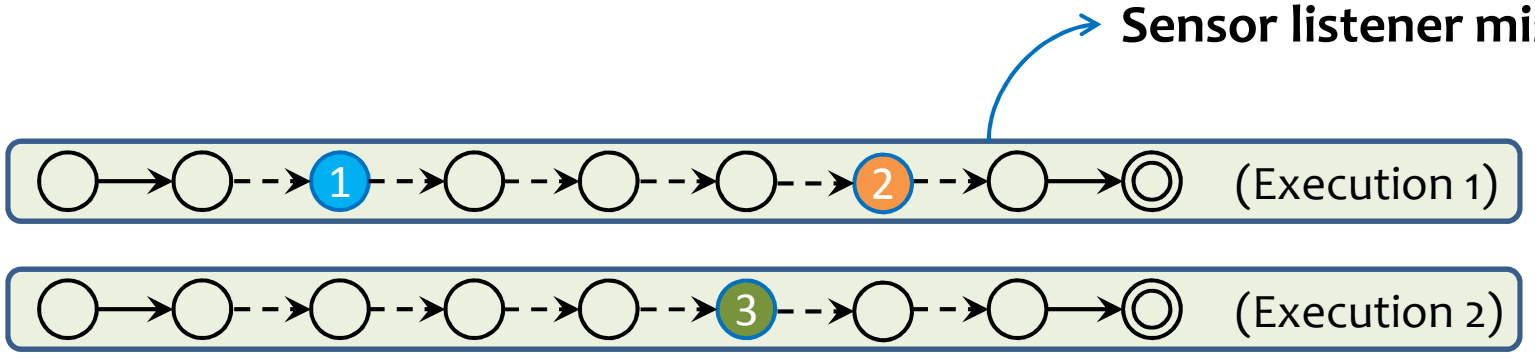
Usage Comparison

$$utilization\_coefficient(s,d) = \frac{usage(s,d)}{Max_{s' \in S, d' \in D}(usage(s',d'))}$$

Index	Usage	Utilization coefficient
State 1	3n	0.75
State 2	**2n**	**0.5**
State 3	4n	1

Report

- Event sequence
- Sensory data usage details



Evaluation

- RQ1 (Effectiveness): Can GreenDroid effectively detect energy problems?

- RQ2 (Efficiency): How much overhead does GreenDroid incur? Is GreenDroid practical enough to handler real-world large subjects?

Subjects

Application	Basic Information			
	Revision No.	Lines of code	Downloads	Availability
Osmdroid	750	18,091	10K—50K	Google Play
Zmanim	322	4,893	10K—50K	Google Play
Omnidroid	863	12,427	1K—5K	Google Play
DroidAR	204	18,106	1K—5K	Google Code
Recycle-locator	68	3,241	1K—5K	Google Play
GPSLogger	15	659	1K—5K	Google Code

Effectiveness

Energy Problem	Problem type	New problem
OsmDroid Issue 53	*Sensory data underutilization*	*No*
Zmanim Issue 50	*Sensory data underutilization*	*No*
Zmanim Issue 56	*Sensory data underutilization*	*No*
DroidAR Issue 27	*Sensor listener misusage*	*No*
Recycle-Locator Issue 33	*Sensor listener misusage*	*No*
Omnidroid Issue 179	*Sensory data underutilization*	*Yes*
GPSLogger Issue 7	*Sensory data underutilization*	*Yes*

GreenDroid found seven real problems. Five problems are caused by poor sensory data utilization. Two problems are caused by sensor listener misusage.

Effectiveness

Energy Problem	Problem type	New problem
OsmDroid Issue 53	*Sensory data underutilization*	*No*
Zmanim Issue 50	*Sensory data underutilization*	*No*
Zmanim Issue 56	*Sensory data underutilization*	*No*
DroidAR Issue 27	*Sensor listener misusage*	*No*
Recycle-Locator Issue 33	*Sensor listener misusage*	*No*
Omnidroid Issue 179	*Sensory data underutilization*	*Yes*
GPSLogger Issue 7	*Sensory data underutilization*	*Yes*

First five problems were confirmed before our experiments.
The Last two were new problems found by GreenDroid

Effectiveness

Energy Problem	Problem type	New problem
OsmDroid Issue 53	*Sensory data underutilization*	*No*
Zmanim Issue 50	*Sensory data underutilization*	*No*
Zmanim Issue 56	*Sensory data underutilization*	*No*
DroidAR Issue 27	*Sensor listener misusage*	*No*
Recycle-Locator Issue 33	*Sensor listener misusage*	*No*
Omnidroid Issue 179	*Sensory data underutilization*	*Yes*
GPSLogger Issue 7	*Sensory data underutilization*	*Yes*

"Completely true, Omnidroid does suck up way more energy than necessary. I'd be happy to accept a patch in this regard".

(Omnidroid issue 179)

Efficiency

Application	Analysis Overhead		
	Explored states	Time (seconds)	Space (MB)
Osmdroid	120,189	151	591
Zmanim	54,270	110	205
Omnidroid (12 KLOC)	52,805	220	342
DroidAR (18 KLOC)	91,170	276	217
Recycle-locator	114,709	43	153
GPSLogger	58,824	35	149

Thousands of states explored in a few minutes. Memory Consumption well supported by modern PCs even without optimization.

Efficiency

Application	Analysis Overhead		
	Explored states	Time (seconds)	Space (MB)
Osmdroid (18 KLOC)	120,189	151	591
Zmanim	54,270	110	205
Omnidroid (12 KLOC)	52,805	220	342
DroidAR (18 KLOC)	91,170	276	217
Recycle-locator	114,709	43	153
GPSLogger	58,824	35	149

Large subjects' analysis overhead suggests that GreenDroid is practical enough to handle real world Android applications.

Discussion

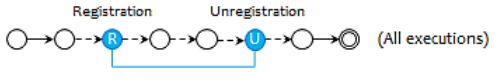
- Limitations
  - Imprecision in AEM model and native lib modeling
  - Complex inputs generation (e.g., password)
  - Limited subjects in evaluation

- Future work
  - Validate the effectiveness with more subjects
  - Investigate energy problems caused by other reasons (e.g., network)

Conclusion

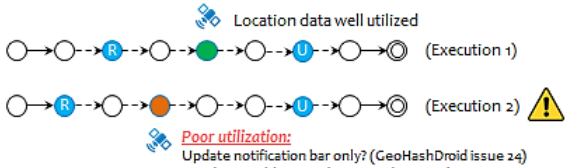
Patterns

- Sensor listener misusage



"Always un-register sensor listener before program exits!"

- Sensory data underutilization

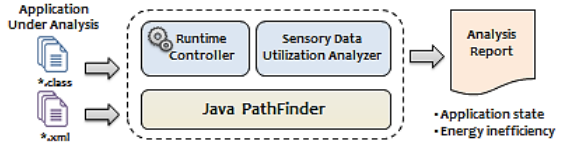


Poor utilization:
Update notification bar only? (GeoHashDroid issue 24)
Render invisible map? (OsmDroid issue 53)

9/20

Approach Overview

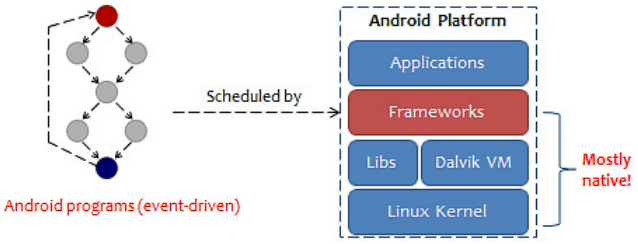
- Dynamic analysis (on top of Java PathFinder)
- Simple, scalable, and effective



11/20

App execution & state exploration in JPF

- Absence of explicit control flow (event-driven)
- Heavy reliance on native system libs (platform specific)
- Essentially interactive (valid user input generation)



12/20

Evaluation

- RQ1 (Effectiveness): Can we approach effectively detect energy inefficiency problems?
- RQ2 (Efficiency): How much overhead does our approach incur?
- RQ3 (Comparison): How does our approach compare with existing resource leak detection techniques?

23/20

Thank you!