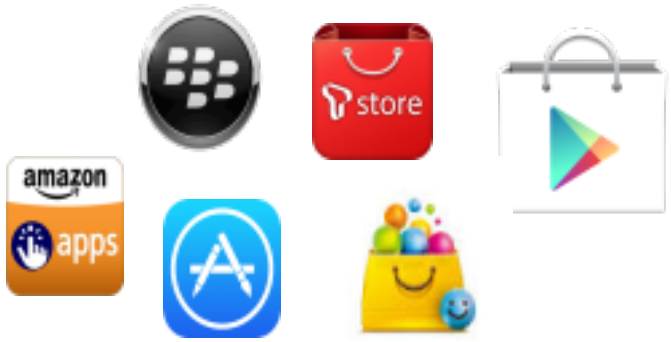
APPSTORE 2.0: *Improving the Quality of Mobile Apps by Leveraging the Crowds*


iPhone 1st Generation
iPhone 3G
iPhone 3GS
iPhone 4
iPhone 4S
iPhone 5
iPhone 5C
iPhone 5S
iPhone 6
iPhone 6 Plus

Motivation

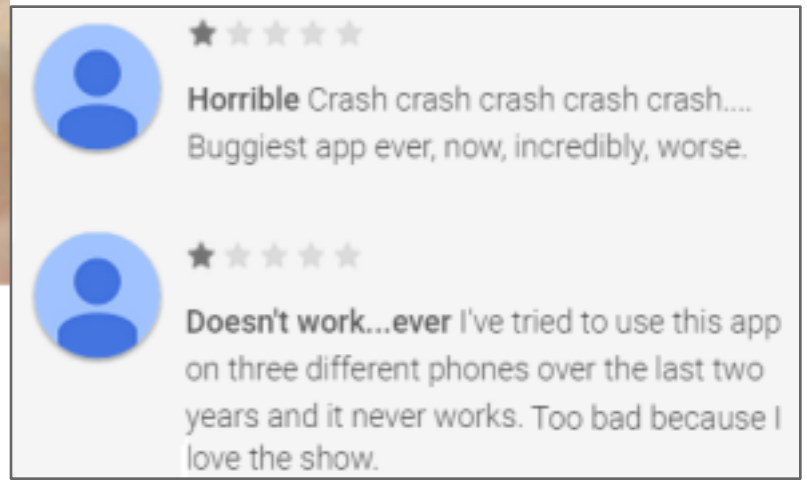
Growing mobile app stores



In 2013, Google Play Store
+2,000,000 apps

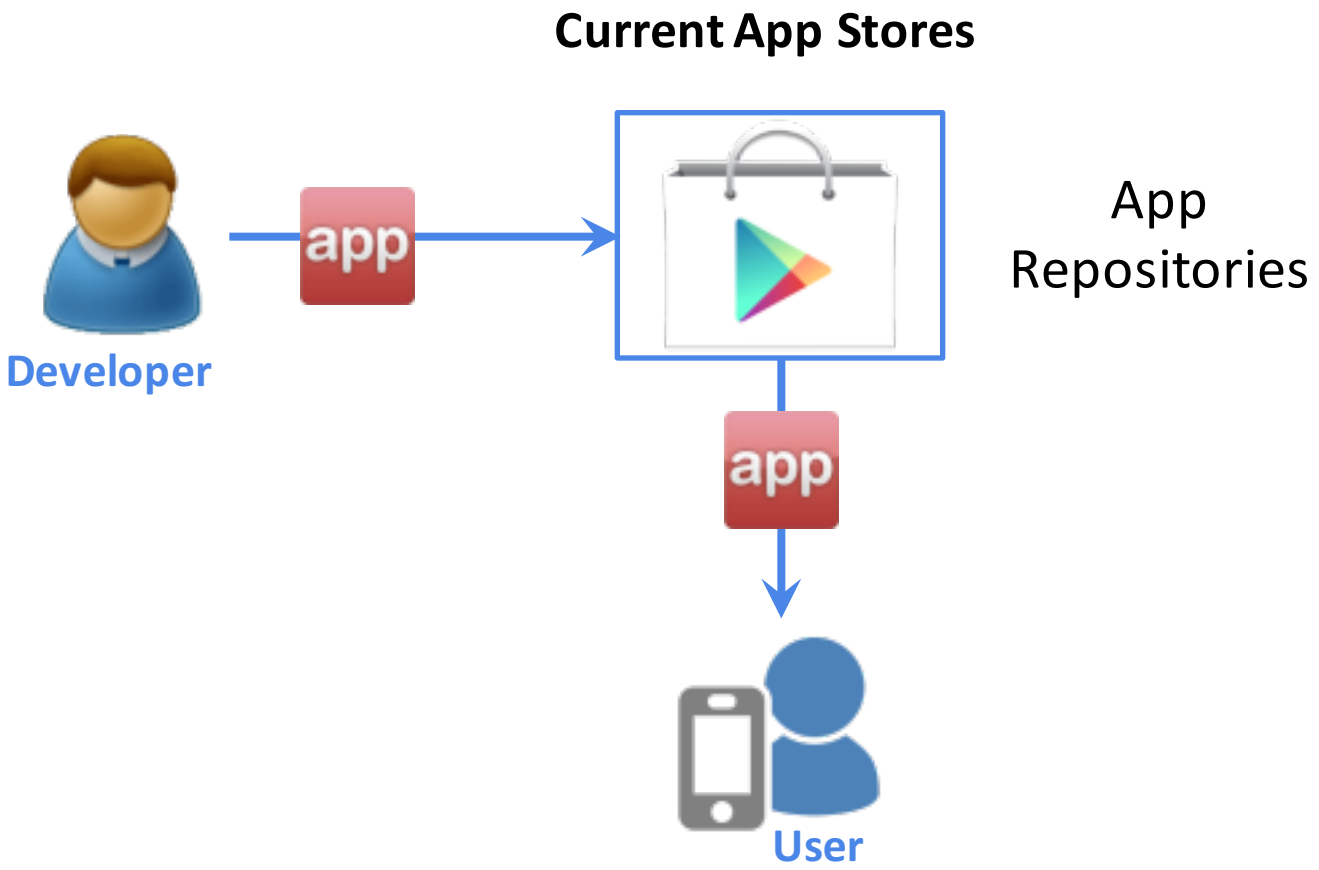
+50 billions
app downloads!

Motivation

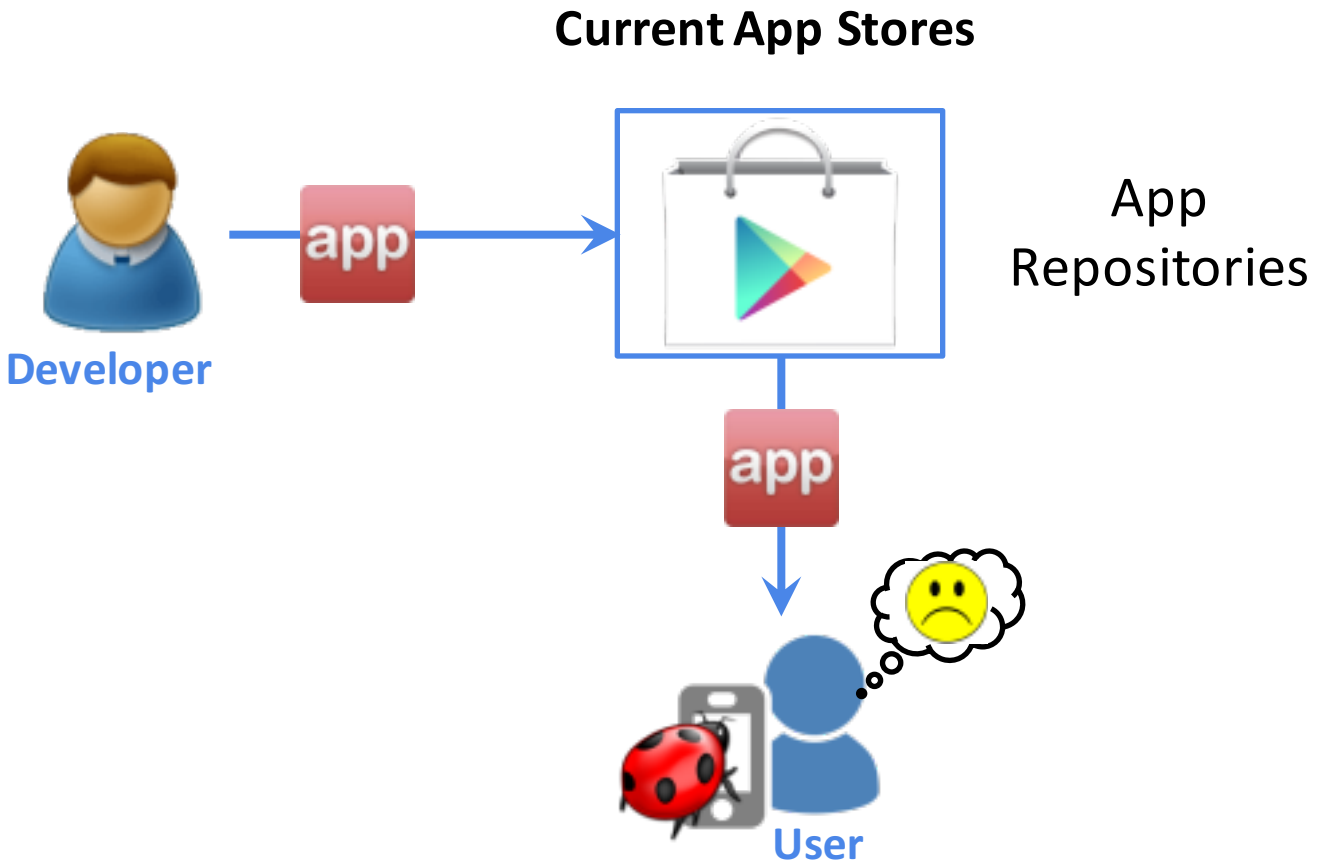


Negative impact on developer and store reputation!

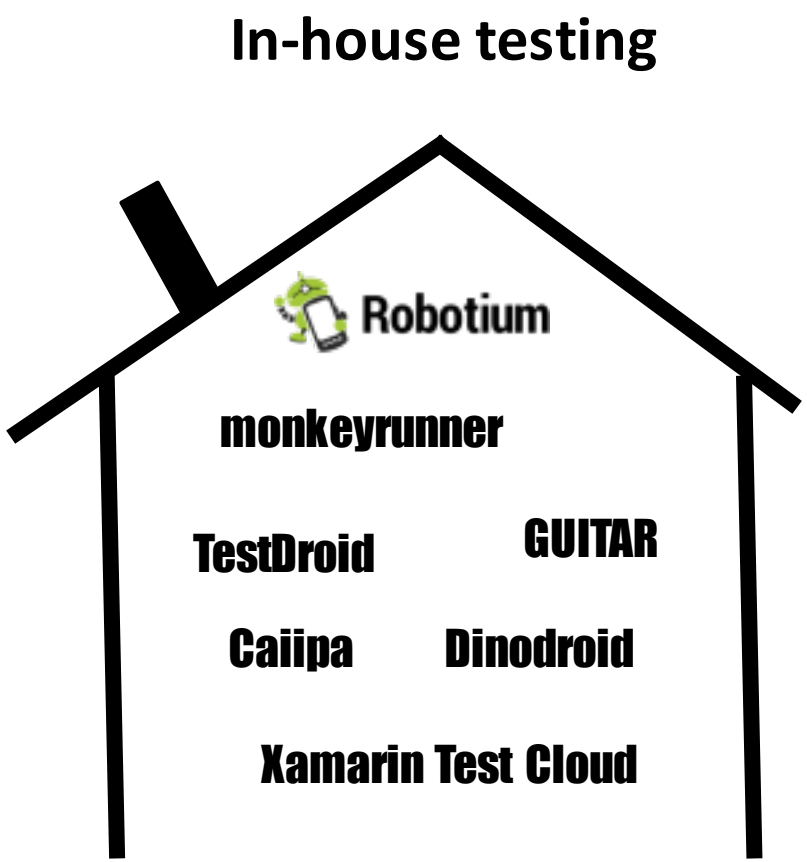
Motivation


Current App Stores
app
app
Developer
User
App
Repositories

Motivation

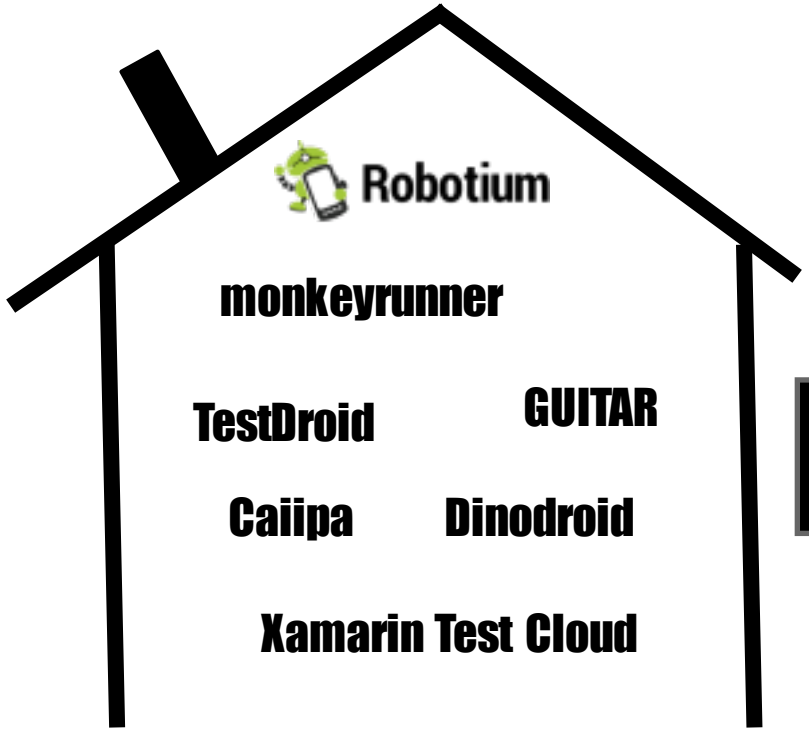

Current App Stores
app
Developer
App
Repositories
app
User

Problem


In-house testing
Robotium
monkeyrunner
TestDroid
GUITAR
Caiipa
Dinodroid
Xamarin Test Cloud

Problem

In-house testing



In the wild

Problem

Diagnosing mobile apps is **challenging**

Rapid platform evolution

Device fragmentation

Diverse operating context

Privacy issues

Leveraging Crowds of Apps to Increase Quality

Geoffrey Hecht,
Naouel Moha (UQAM)


Memos
iPod
Notes
Boxee
Bandsintown
Weather
Echofon
Facebook
iTunes
CameraBag
Buamai
TED
Camera
Byline
Contacts
Bebot
Photos
App Store
NetNewsWire
Showtimes
Hipstamatic
Clock
Calendar
Tweetie
Opera Mini
QuickMark
Air Mouse
Shopper
tvChatter
Photos
App Store
NetNewsWire
Showtimes
Hipstamatic
Clock
Calendar
Tweetie
Settings
YouTube
Compass
Ruler
Cameraman
CameraBag
App Store
MiniPiano
Easy Wi-Fi
Jaadu VNC
chesspuzzles
Air Mouse
iMapMyRide
Mobile News
Bloomberg
N
Urbanspoon
Labyrinth LE
Ruler
Cameraman
CameraBag
App Store
MiniPiano
Easy Wi-Fi
Jaadu VNC
chesspuzzles
Air Mouse
Twitterrific
Chess
Gizmodo
Remote
Reader
NPR
Facebook
MySpace
Contacts
Yelp
Constitution
GraphCalc
AirSharing
Yammer
Pageonce
IKTL
::phot...hic::
WordPress
Calculator
Gizmodo
Remote
Reader
NPR
Facebook
MySpace
Contacts
Yelp
Constitution
GraphCalc
iMapMyRide
Showtimes
Grends
Wells Fargo
MindWave
Stocks
PSFK...PS
iDesigner
Urbanspoon
TheAleatory
AirSharing
Yammer

Development AntiPattern:

The Blob

- **Symptoms**
  - Single class with many attributes & operations
  - Controller class with simple, data-object classes.
  - Lack of OO design.
  - A migrated legacy design
- **Consequences**
  - Lost OO advantage
  - Too complex to reuse or test.
  - Expensive to load

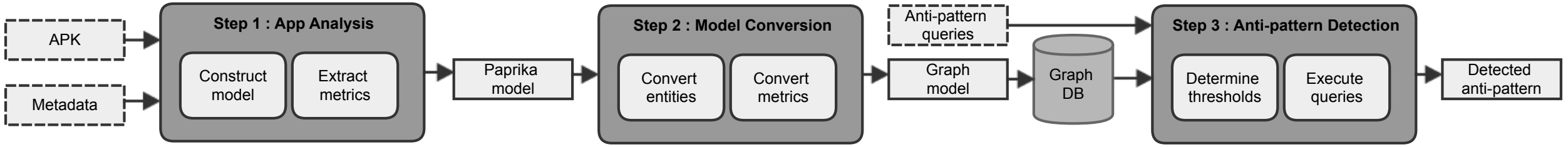
MITRE

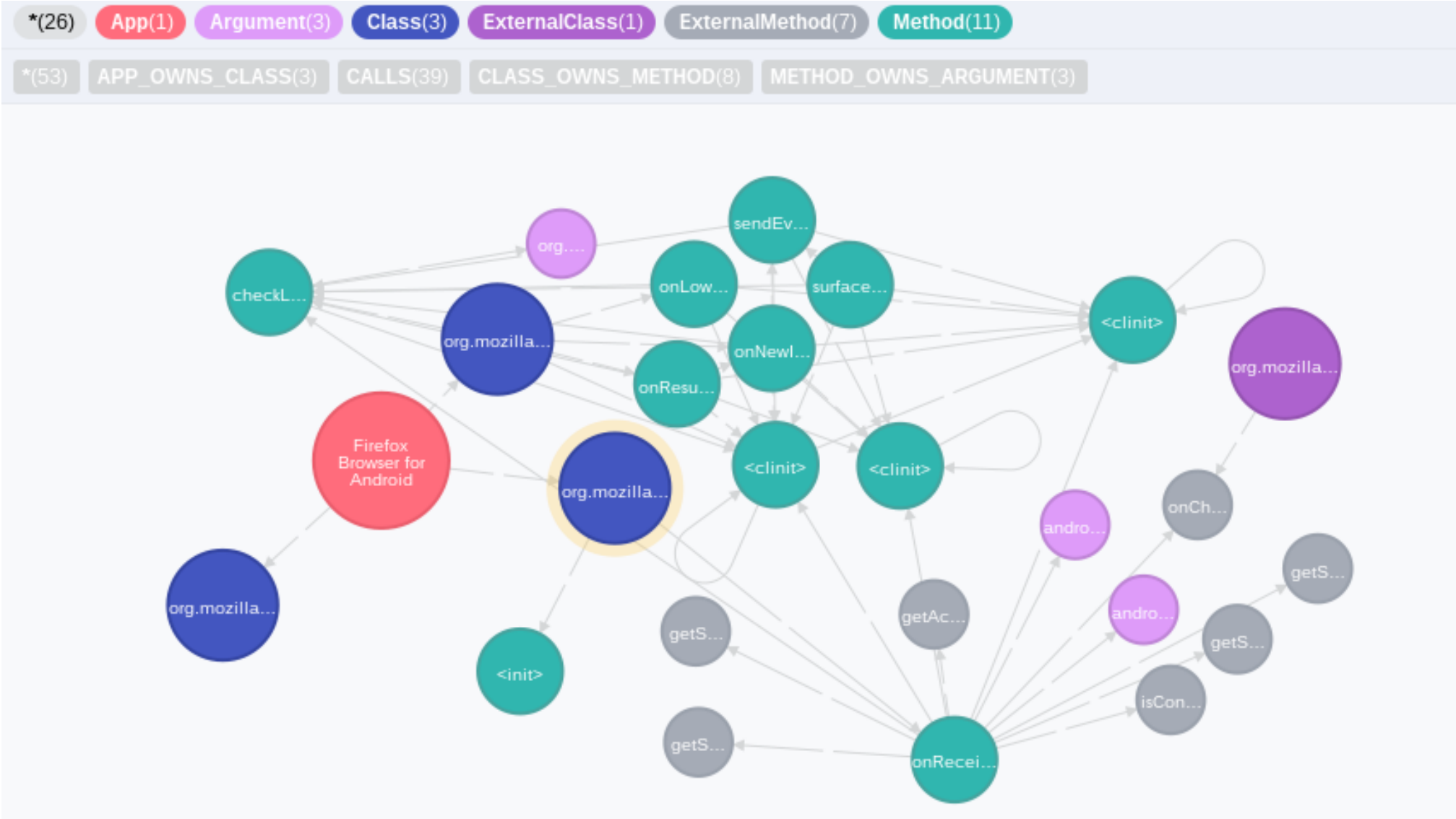
Avoid Internal Getters/Setters

In native languages like C++ it's common practice to use getters (`i = getCount()`) instead of accessing the field directly (`i = mCount`). This is an excellent habit for C++ and is often practiced in other object oriented languages like C# and Java, because the compiler can usually inline the access, and if you need to restrict or debug field access you can add the code at any time.

However, this is a bad idea on Android. Virtual method calls are expensive, much more so than instance field lookups. It's reasonable to follow common object-oriented programming practices and have getters and setters in the public interface, but within a class you should always access fields directly.

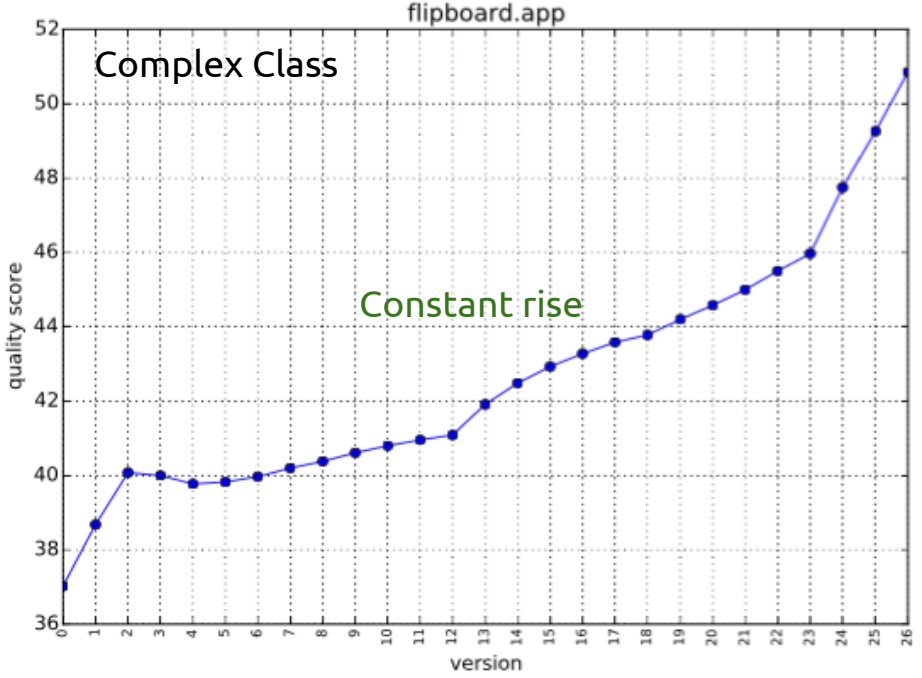
Without a JIT, direct field access is about 3x faster than invoking a trivial getter. With the JIT (where direct field access is as cheap as accessing a local), direct field access is about 7x faster than invoking a trivial getter.

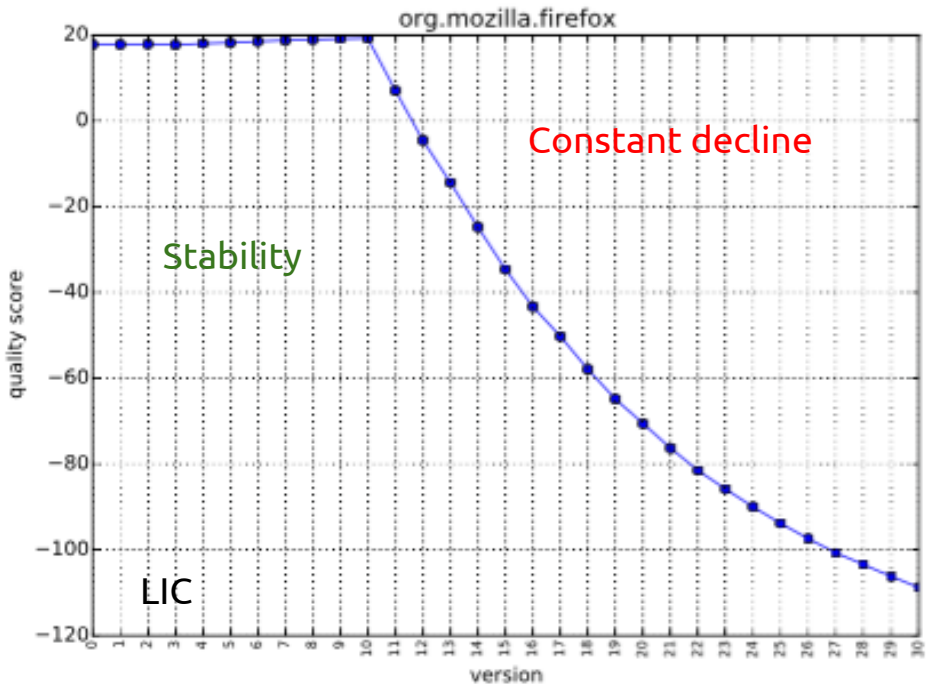

APK
Metadata
Step 1 : App Analysis
Construct model
Extract metrics
Paprika model
Step 2 : Model Conversion
Convert entities
Convert metrics
Graph model
Anti-pattern queries
Graph DB
Step 3 : Anti-pattern Detection
Determine thresholds
Execute queries
Detected anti-pattern

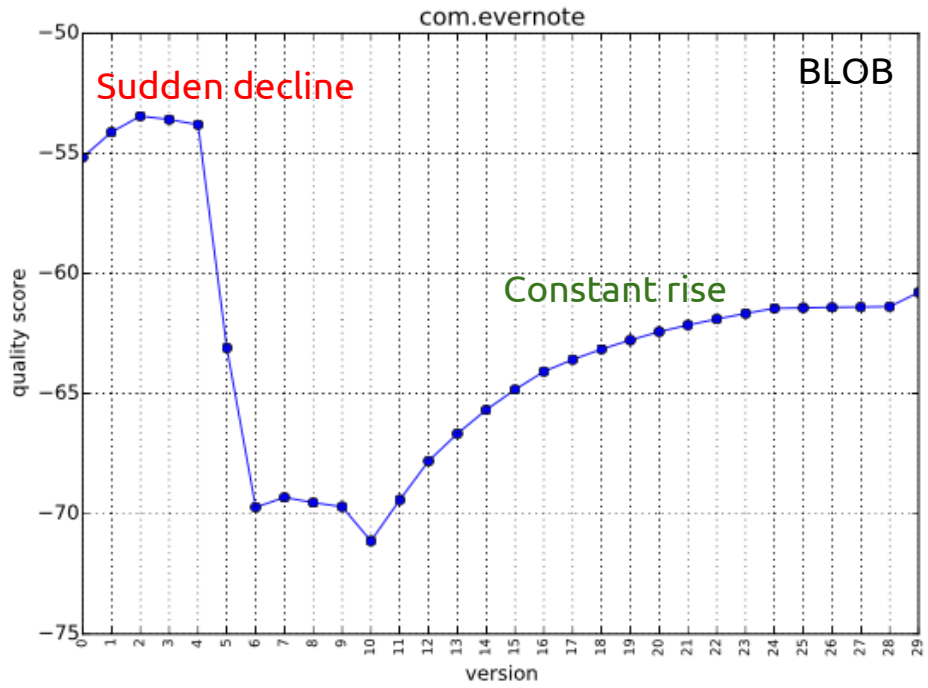

*(26)
App(1)
Argument(3)
Class(3)
ExternalClass(1)
ExternalMethod(7)
Method(11)
*(53)
APP_OWNS_CLASS(3)
CALLS(39)
CLASS_OWNS_METHOD(8)
METHOD_OWNS_ARGUMENT(3)
org....
sendEv...
checkL...
onLow...
surface...
org.mozilla...
onNewI...
<clinit>
org.mozilla...
onResu...
Firefox Browser for Android
<clinit>
<clinit>
org.mozilla...
andro...
onCh...
org.mozilla...
getS...
getAc...
andro...
getS...
<init>
getS...
isCon...
getS...
onRecei...

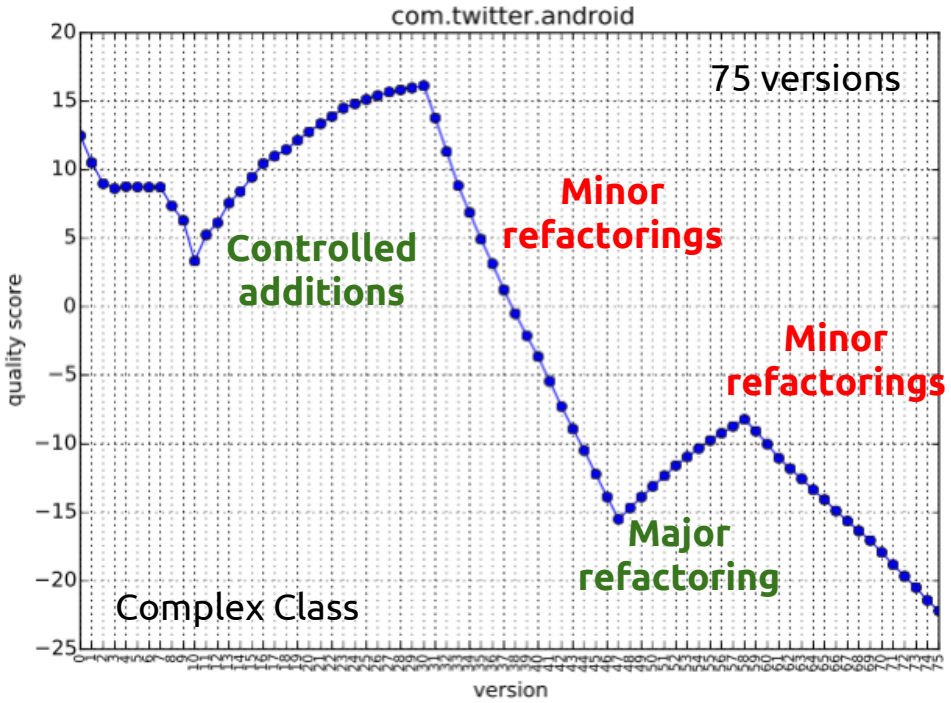
```
MATCH (cl:Class)
WHERE cl.lack_of_cohesion_in_methods > HIGH
  AND cl.number_of_methods > HIGH
  AND cl.number_of_attributes > HIGH
RETURN cl
```

```
MATCH (m1:Method)-[:CALLS]->(m2:Method),
      (cl:Class)
WHERE (m2.is_setter OR m2.is_getter)
  AND cl-[:CLASS_OWNS_METHOD]->m1
  AND cl-[:CLASS_OWNS_METHOD]->m2
RETURN m1
```


flipboard.app
Complex Class
Constant rise
quality score
version


org.mozilla.firefox
Constant decline
Stability
LIC
quality score
version


com.evernote
Sudden decline
BLOB
Constant rise
quality score
version


com.twitter.android
75 versions
Controlled additions
Minor refactorings
Minor refactorings
Major refactoring
Complex Class
quality score
version


Paprika
Android
Neo4J
Blob
Code metrics
3553 apps
17 code smells
7 antipatterns
Graph model
Software architecture
Dexpler
Soot
Skype
Twitter
Facebook
bytecode

Leveraging Crowds of Devices to Improve UI Performance

María Gómez,
Bram Adams (Poly. Montréal)

To ensure smooth interactions apps should run at **60 *Frames per Second*** (fps)

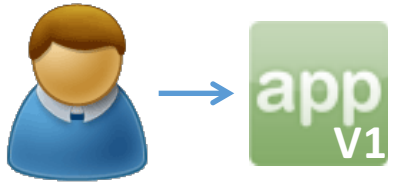
Problem: *How to know when janks arise?*

Nexus 7 – Android **4.3**

Nexus 7 – Android **5.0**

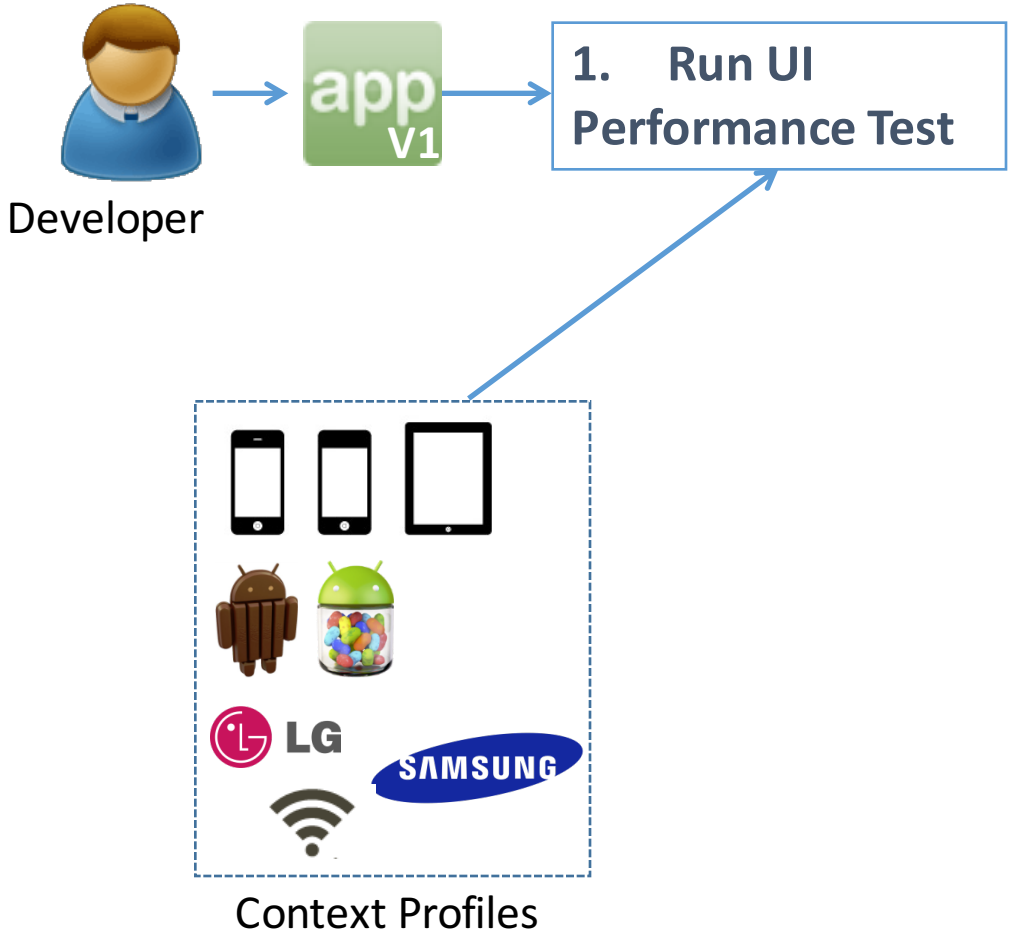
Janky!

DUNE: Identifying Janks

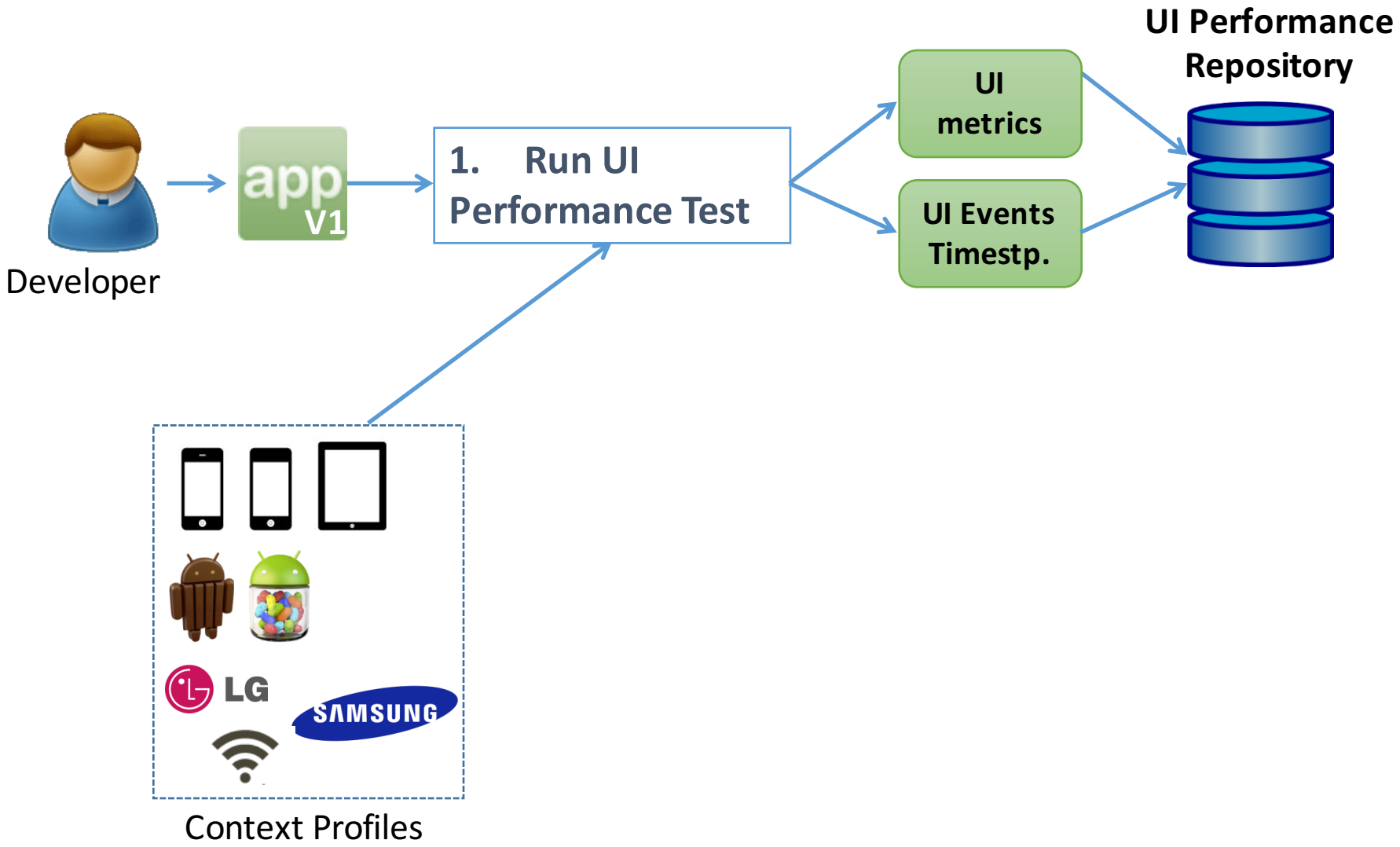


Developer

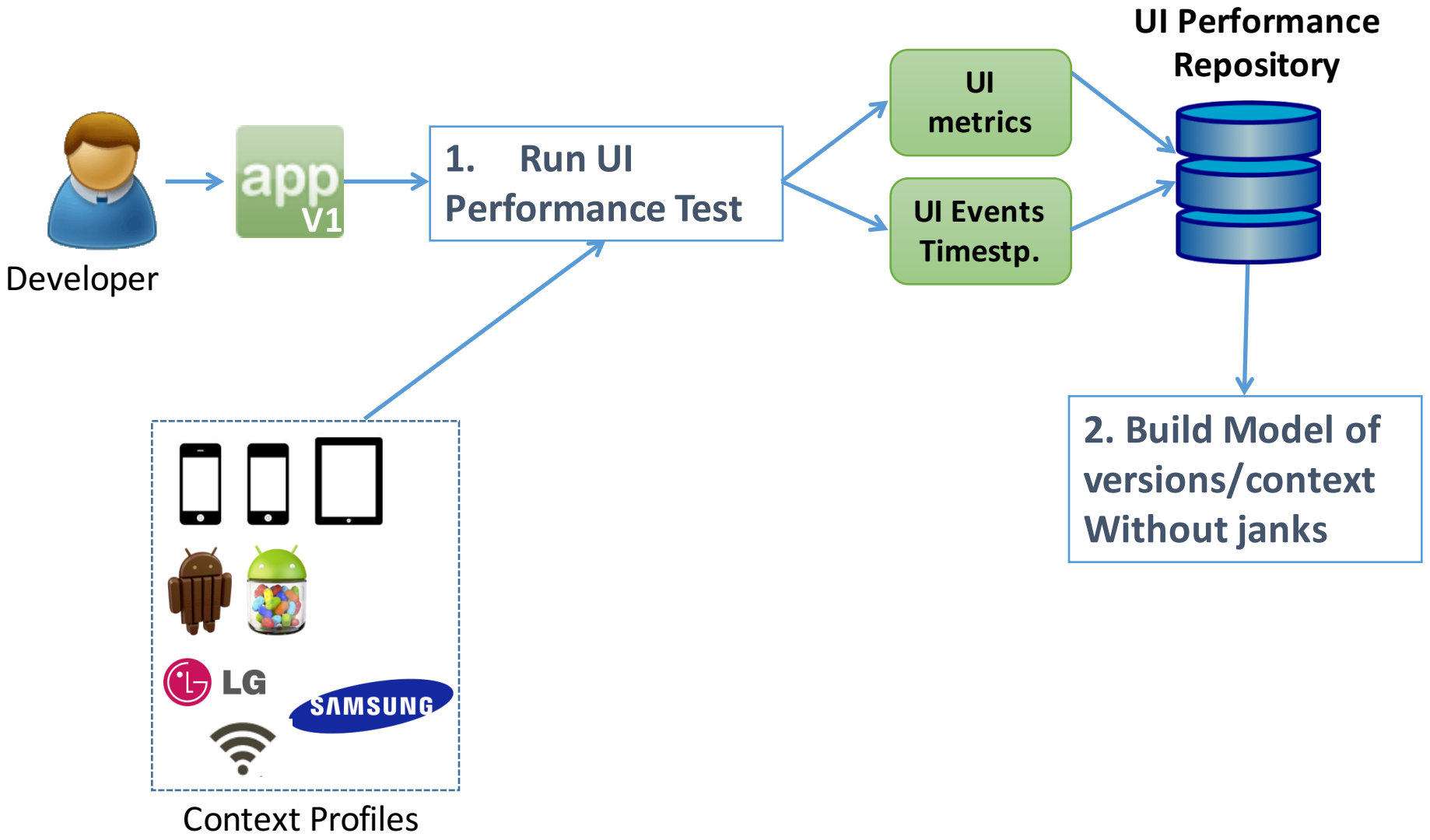
DUNE: Identifying Janks



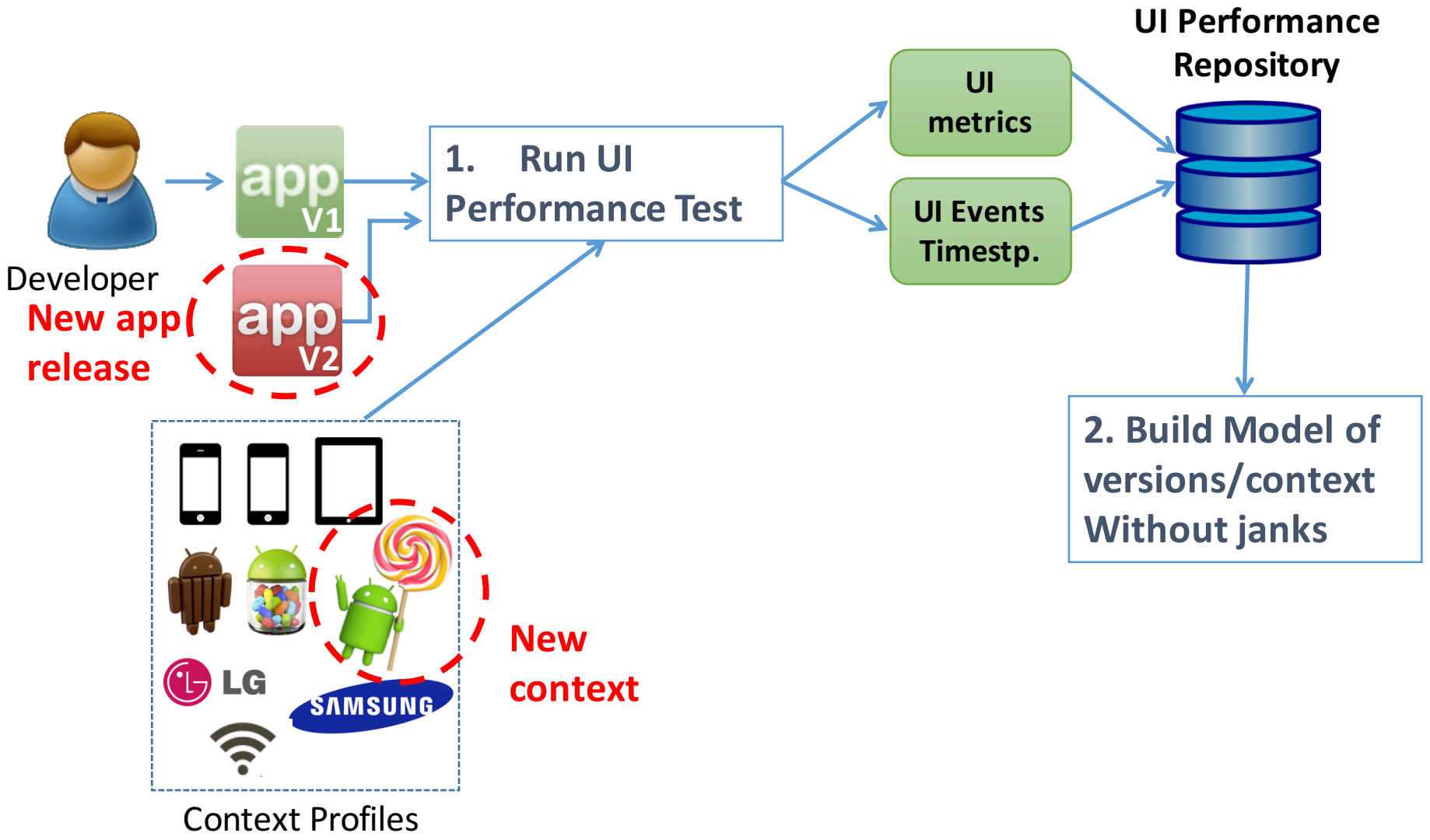
DUNE: Identifying Janks



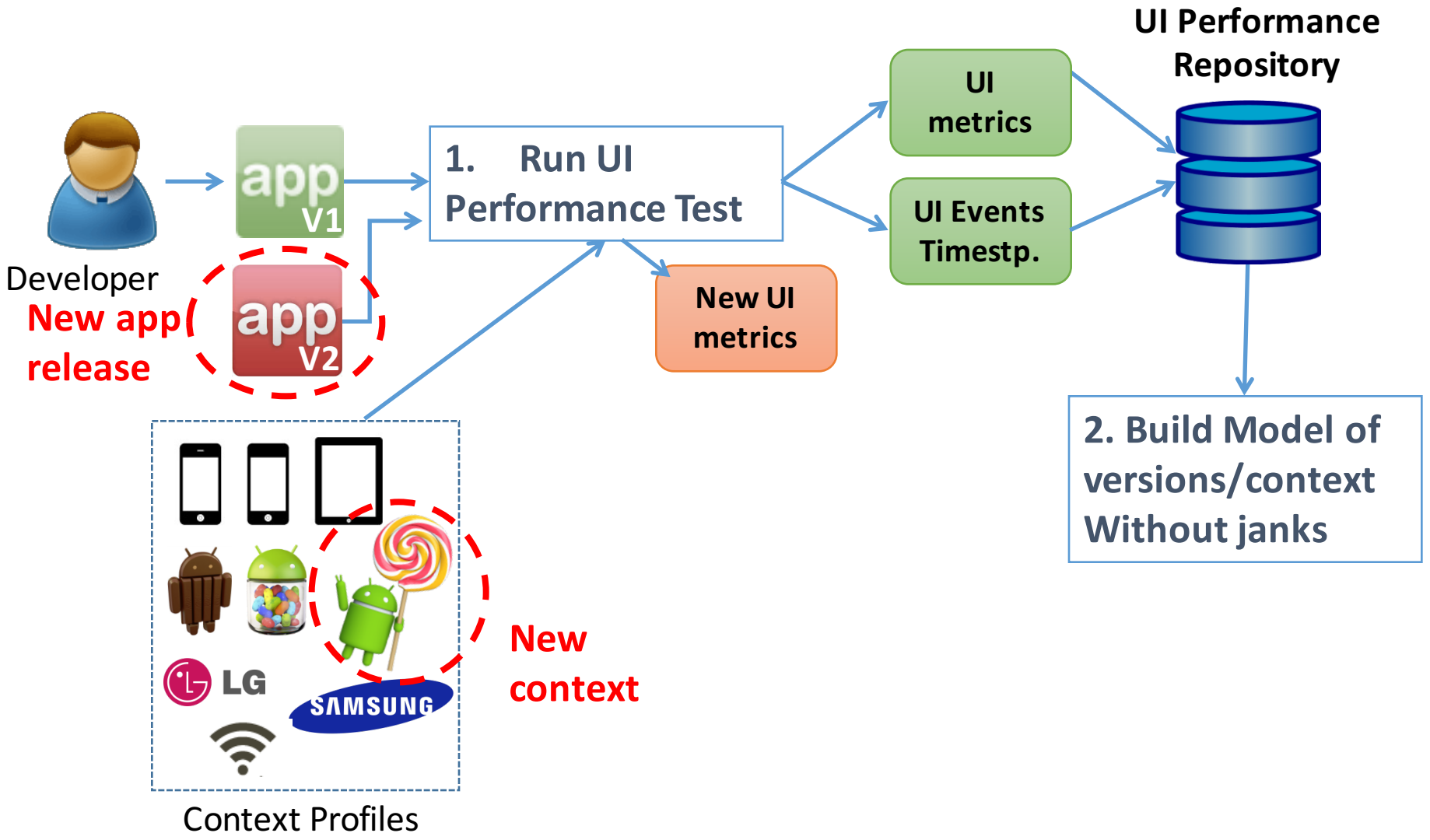
DUNE: Identifying Janks



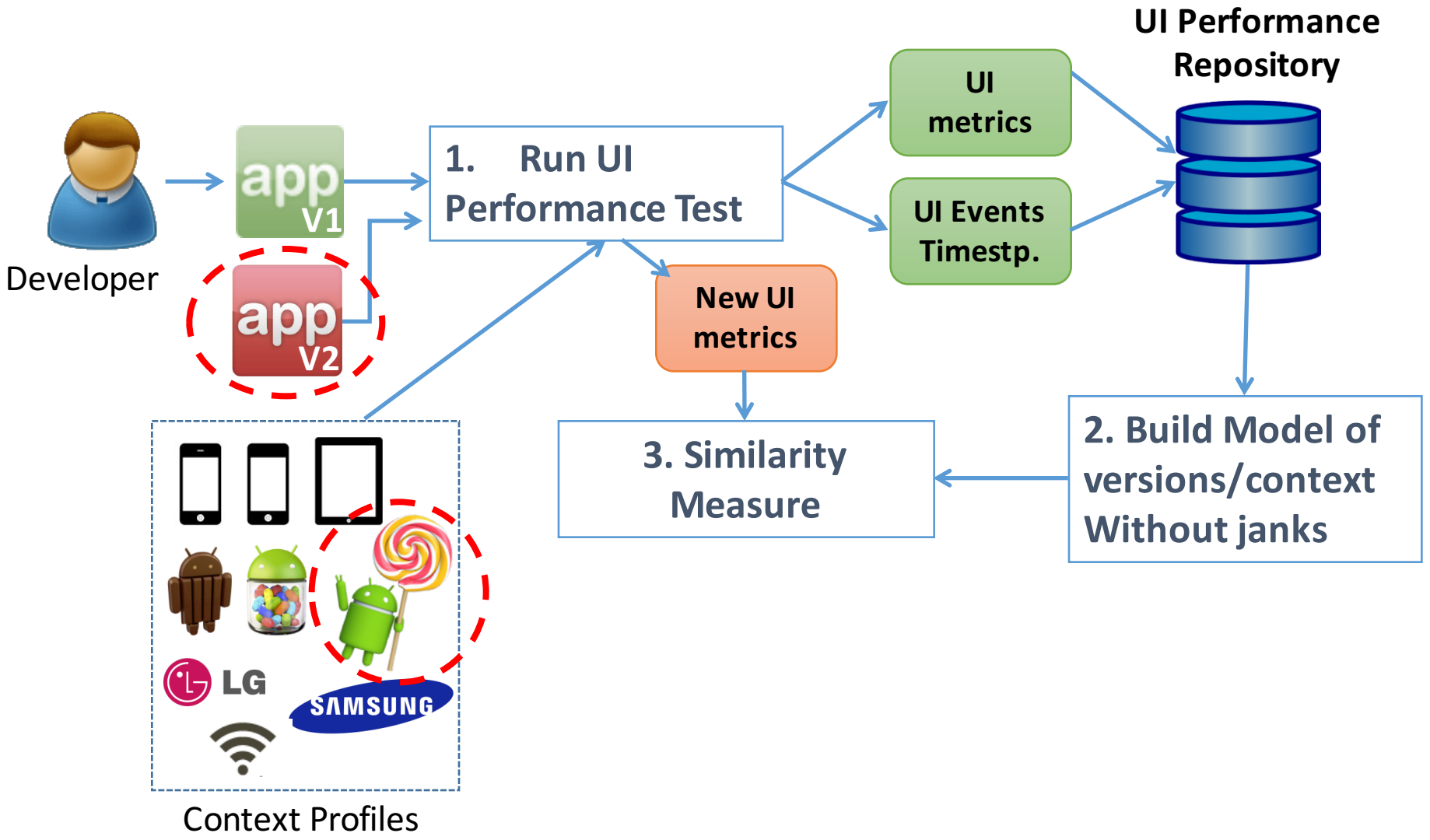
DUNE: Identifying Janks



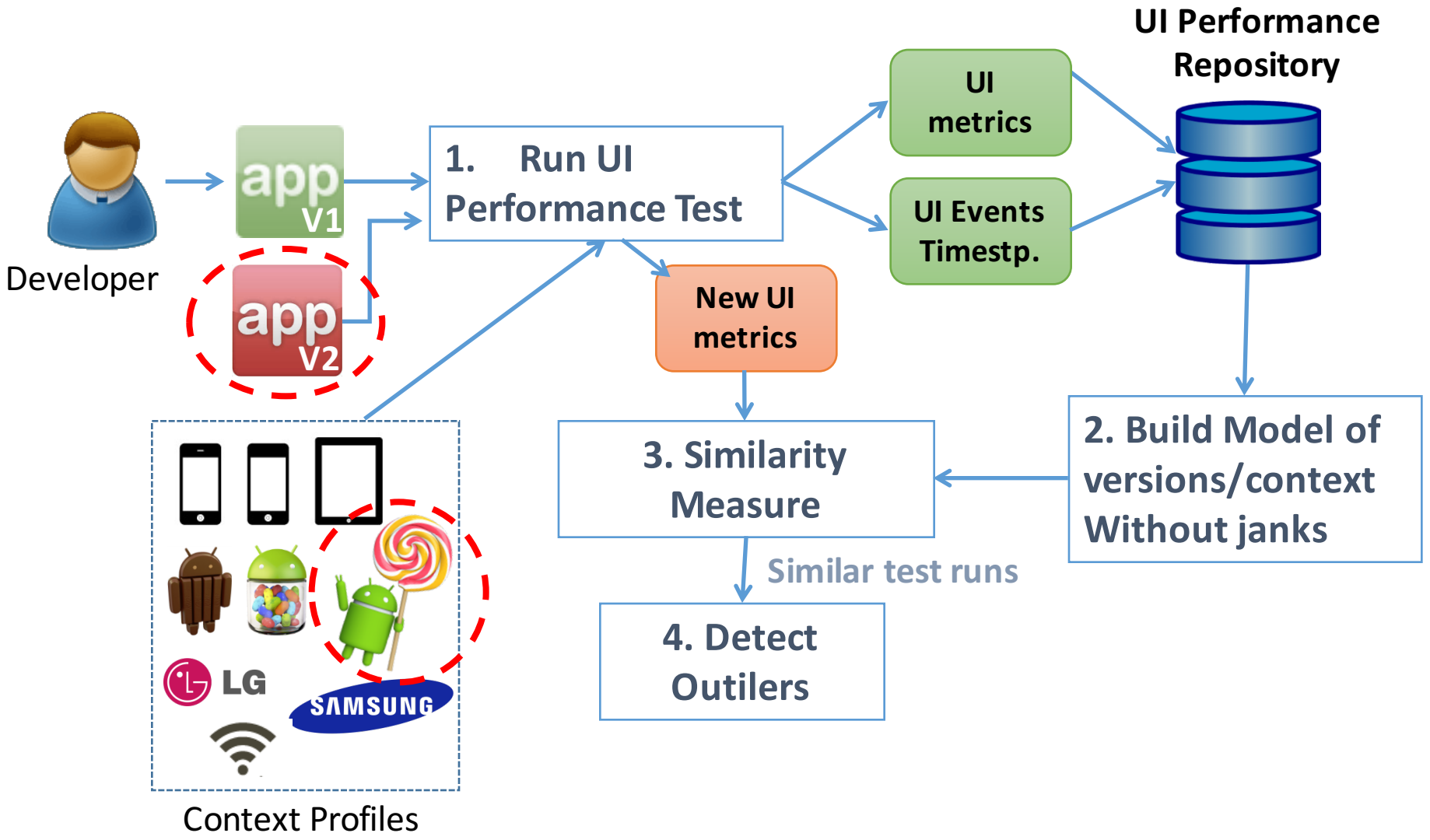
DUNE: Identifying Janks



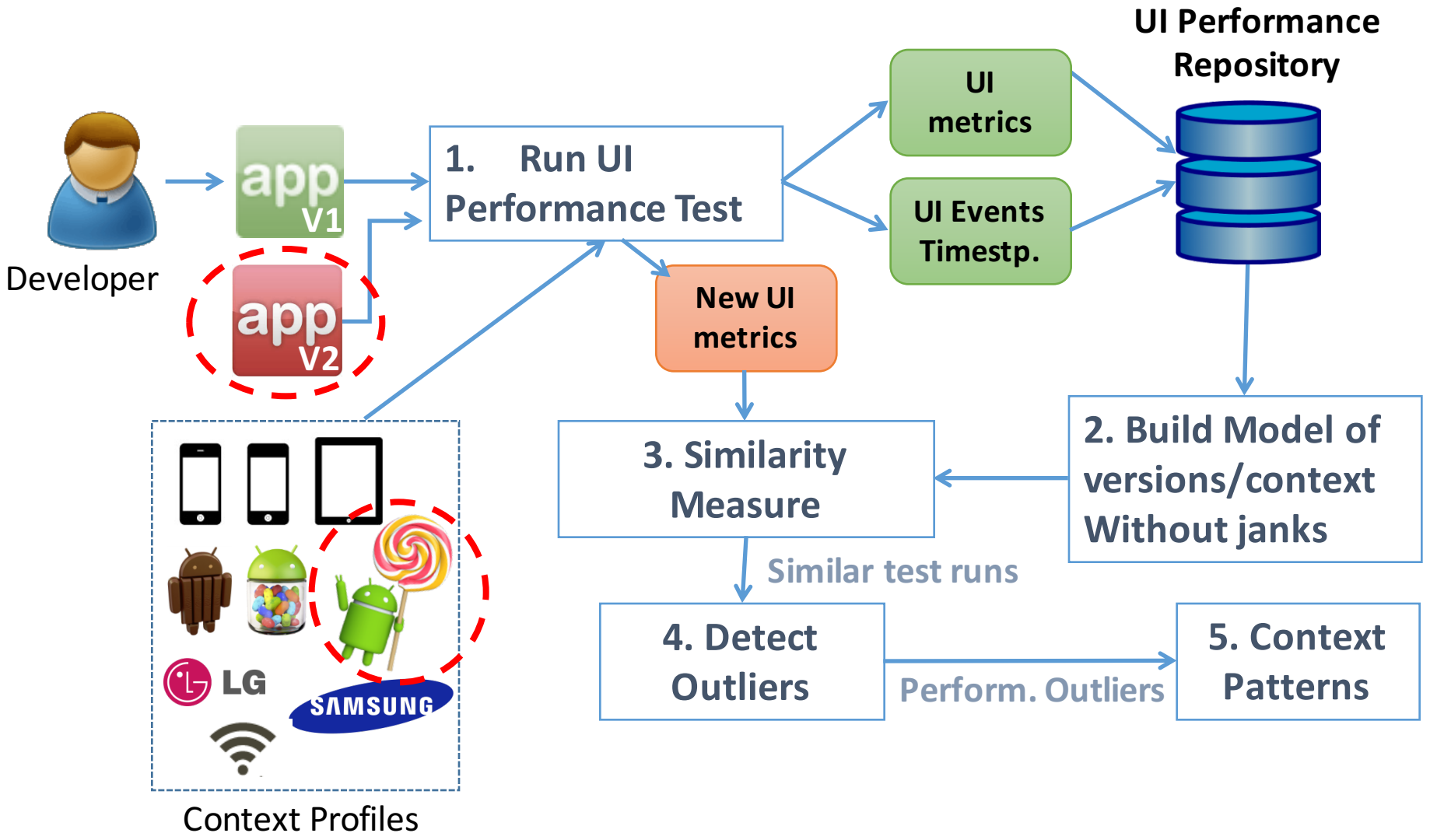
DUNE: Identifying Janks



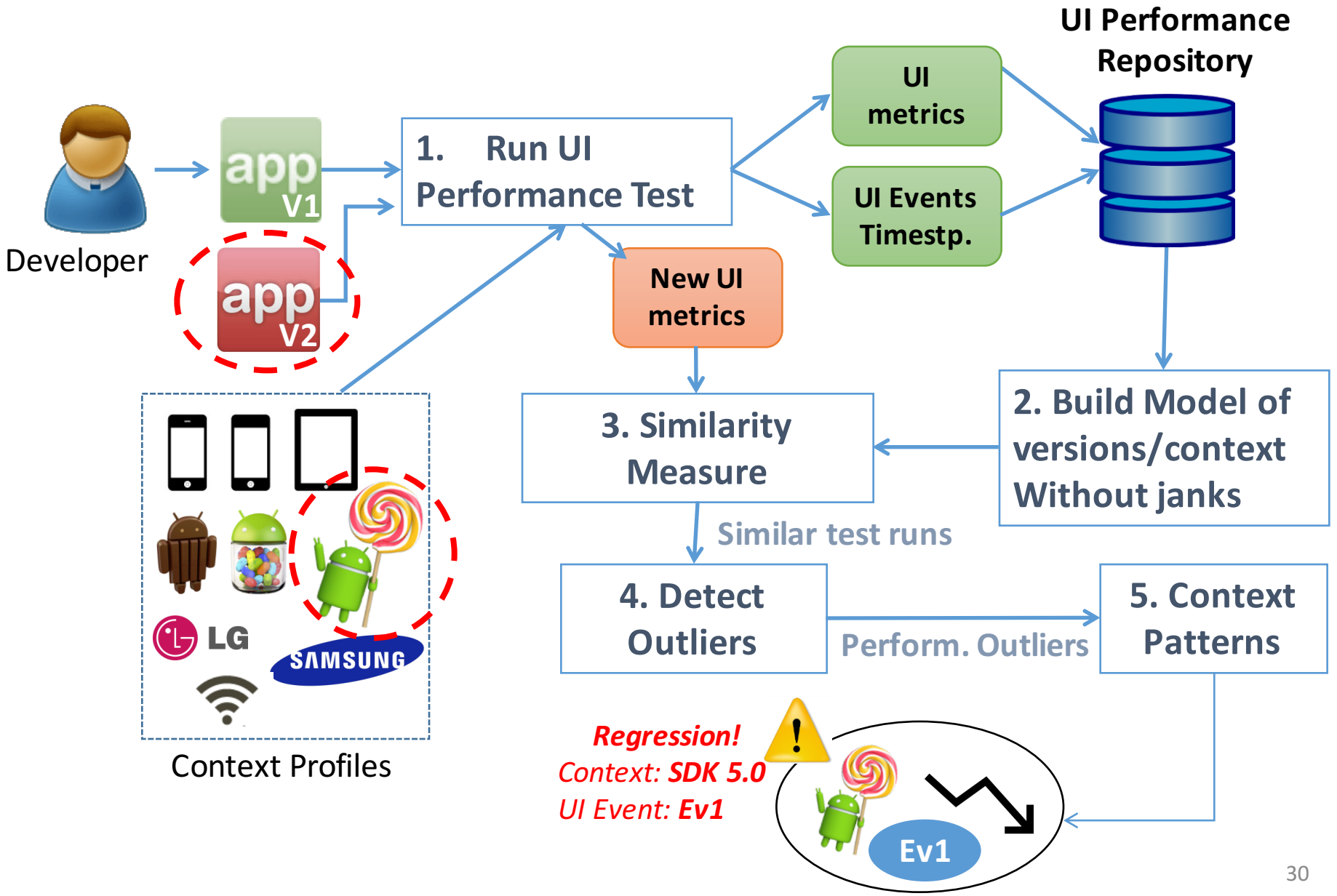
DUNE: Identifying Janks



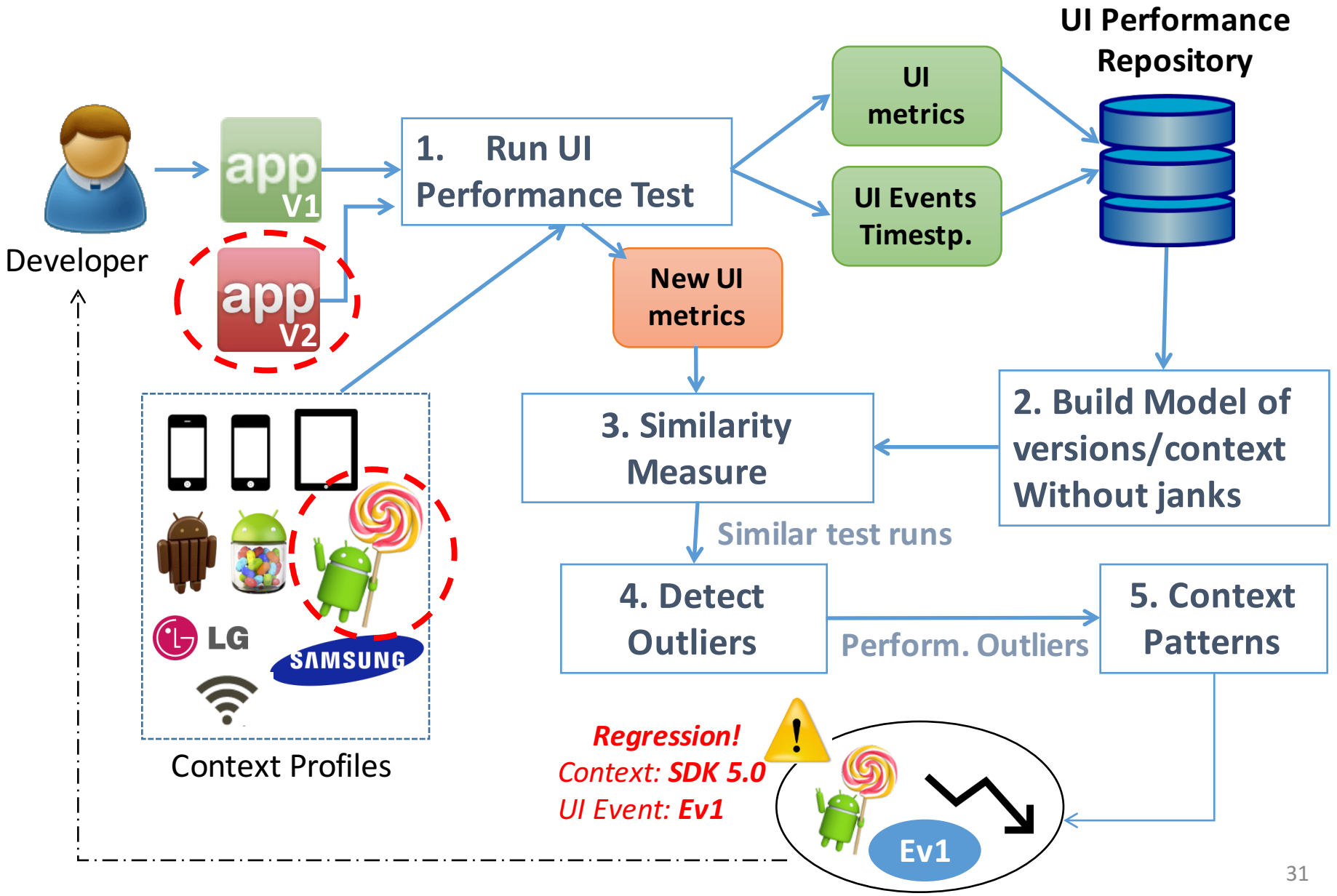
DUNE: Identifying Janks



DUNE: Identifying Janks

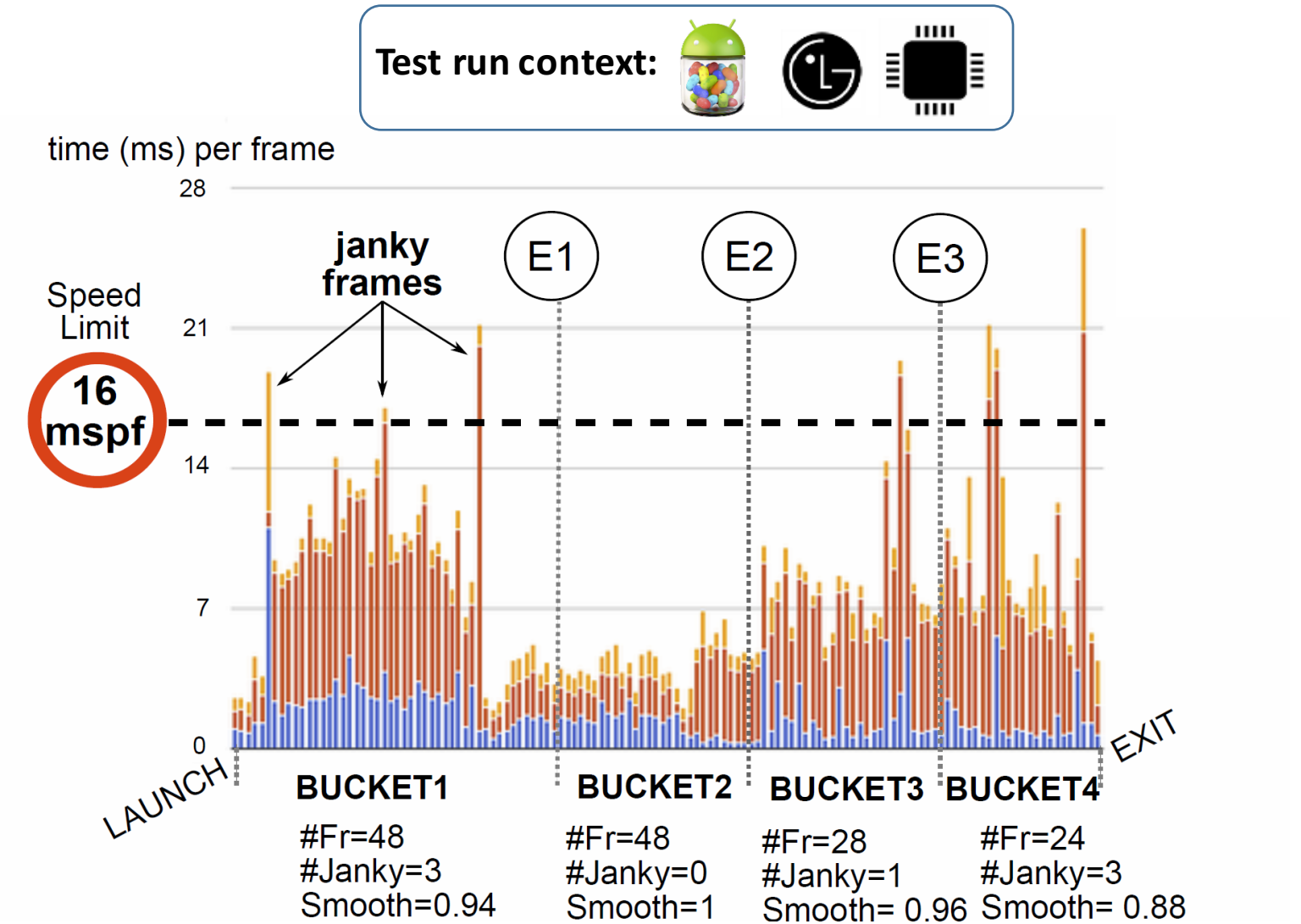


DUNE: Identifying Janks



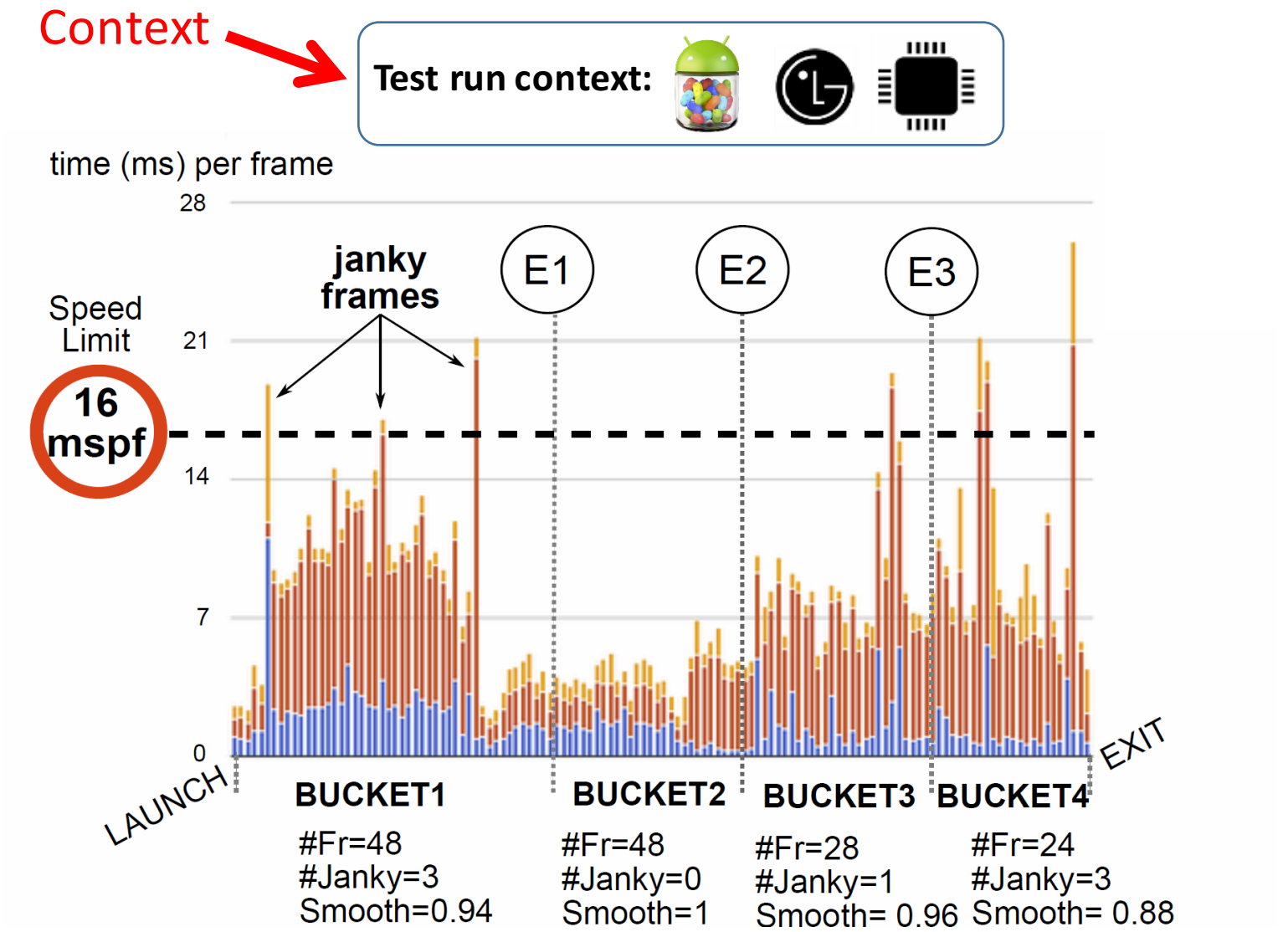
1. Run UI Performance Test

Example. UI metrics collected during a test run



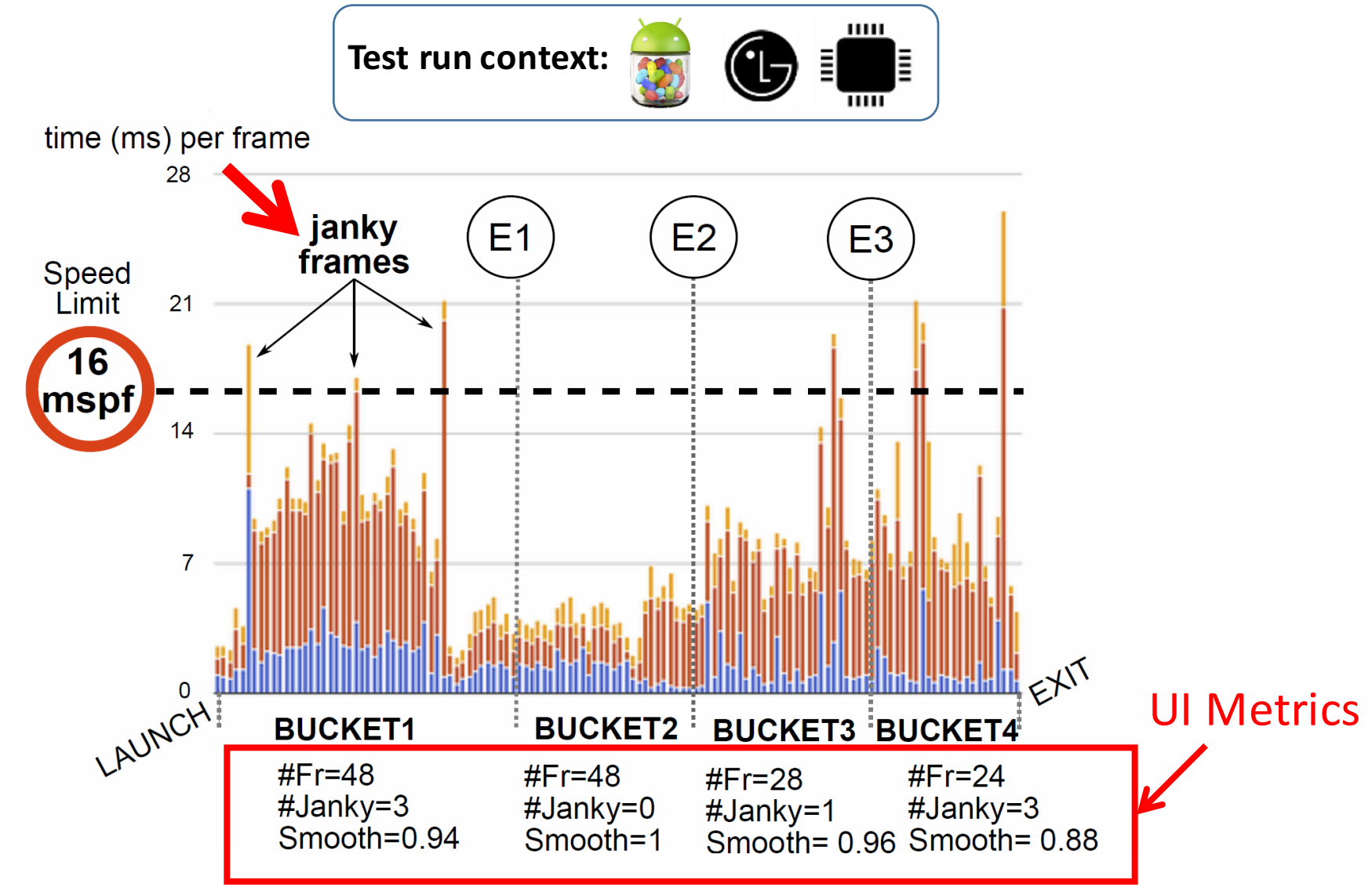
1. Run UI Performance Test

Example. UI metrics collected during a test run



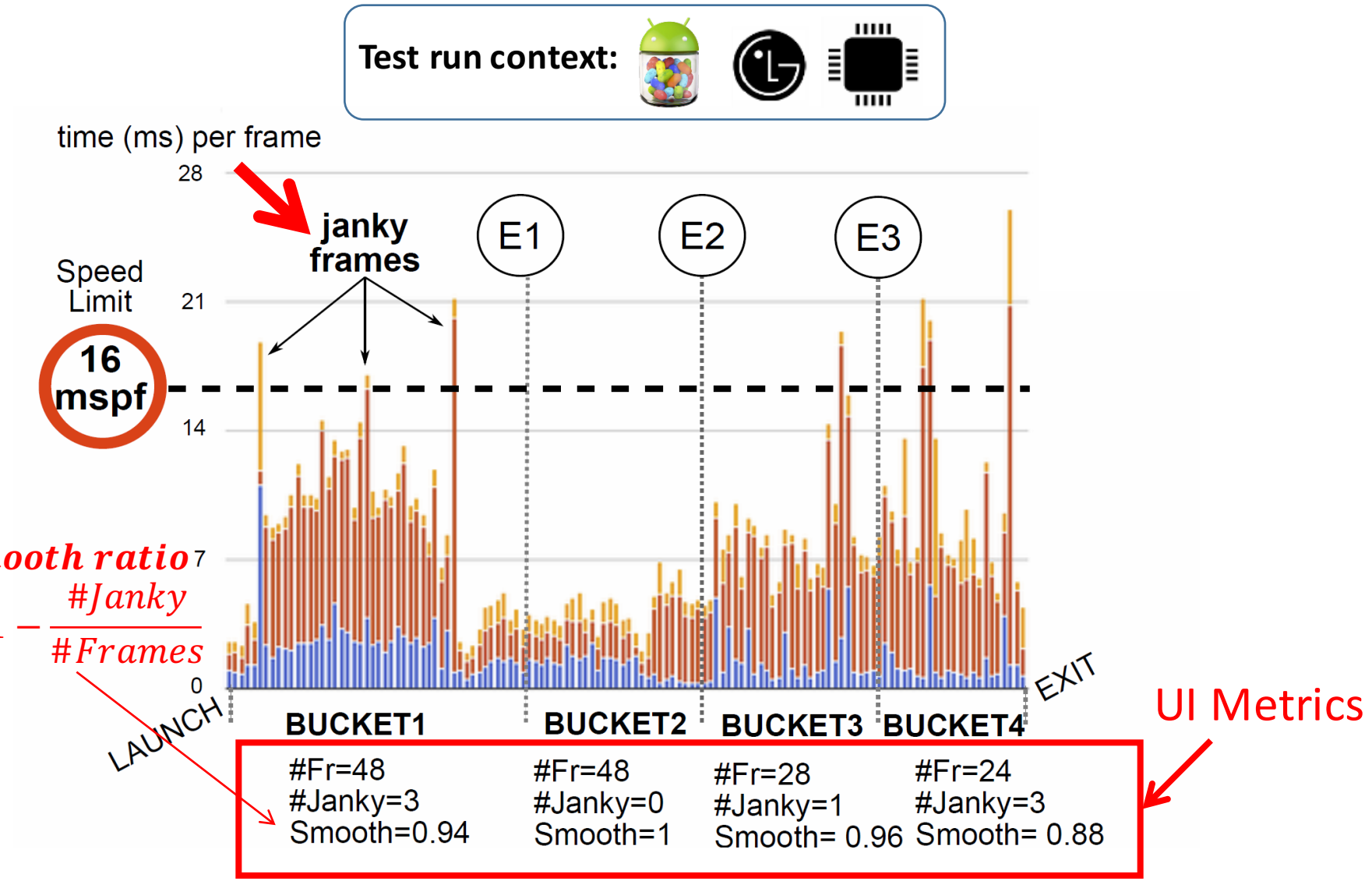
1. Run UI Performance Test

Example. UI metrics collected during a test run



1. Run UI Performance Test

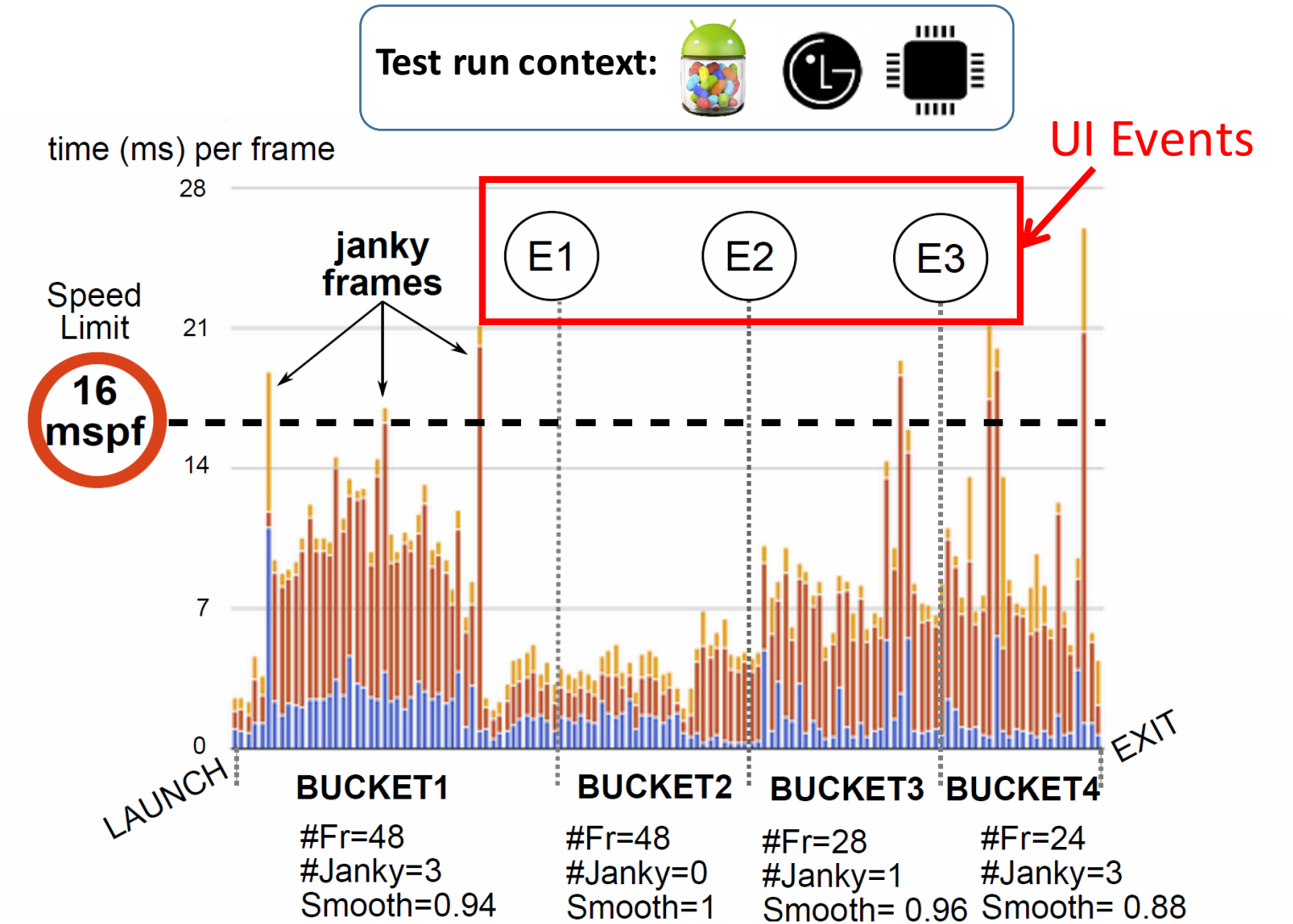
Example. UI metrics collected during a test run



$$\textit{Smooth ratio} = 1 - \frac{\#Janky}{\#Frames}$$

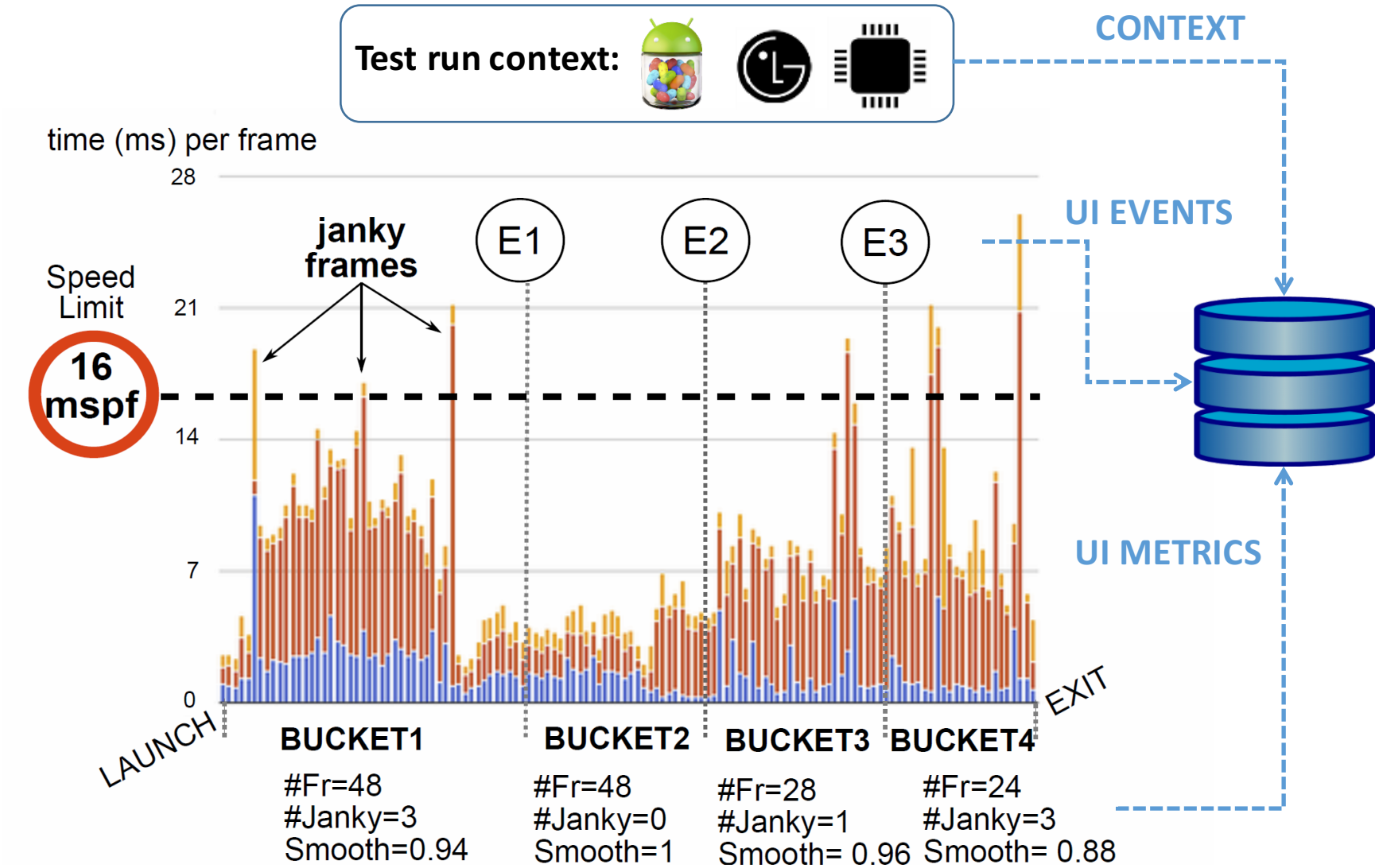
1. Run UI Performance Test

Example. UI metrics collected during a test run



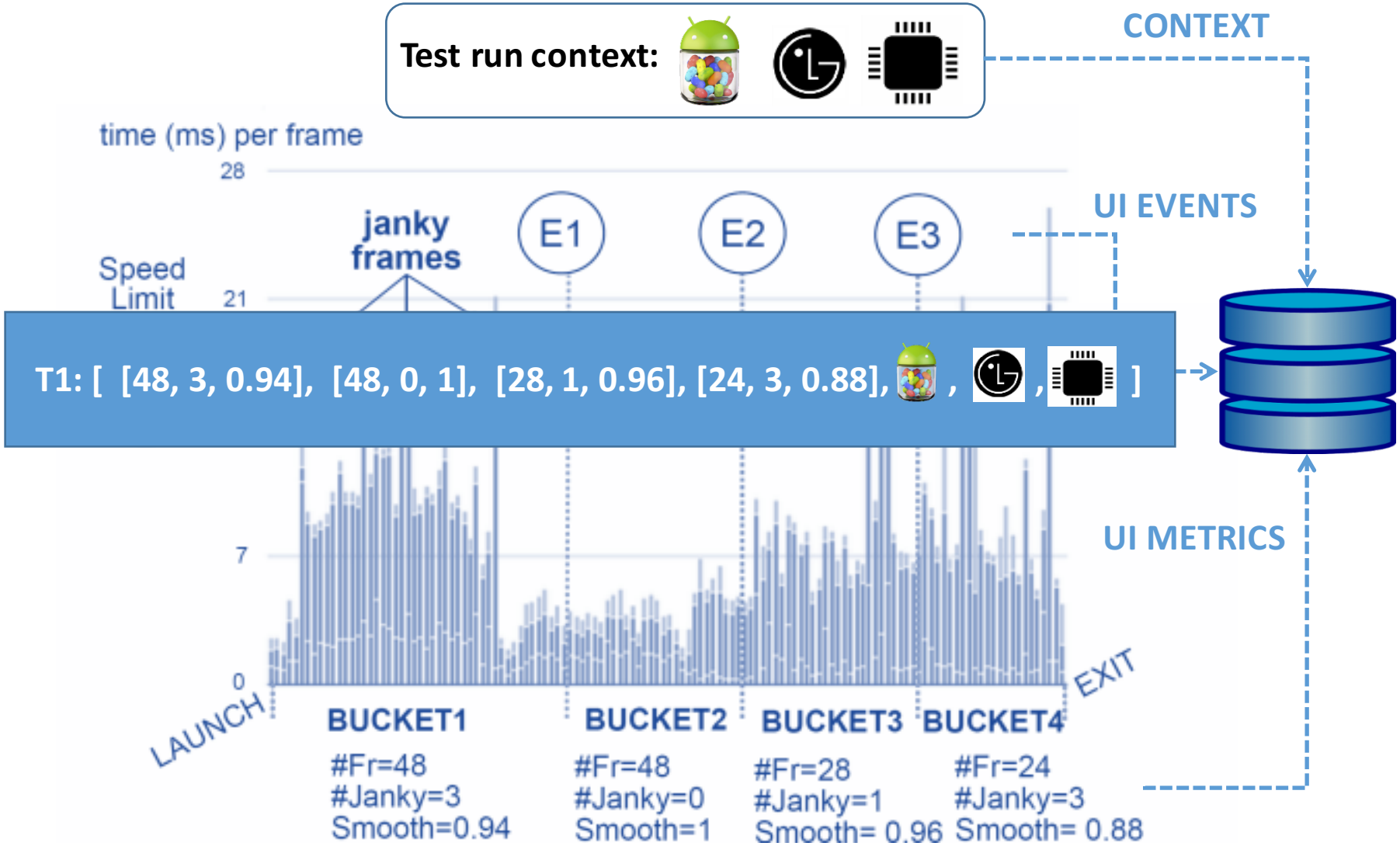
1. Run UI Performance Test

Example. UI metrics collected during a test run



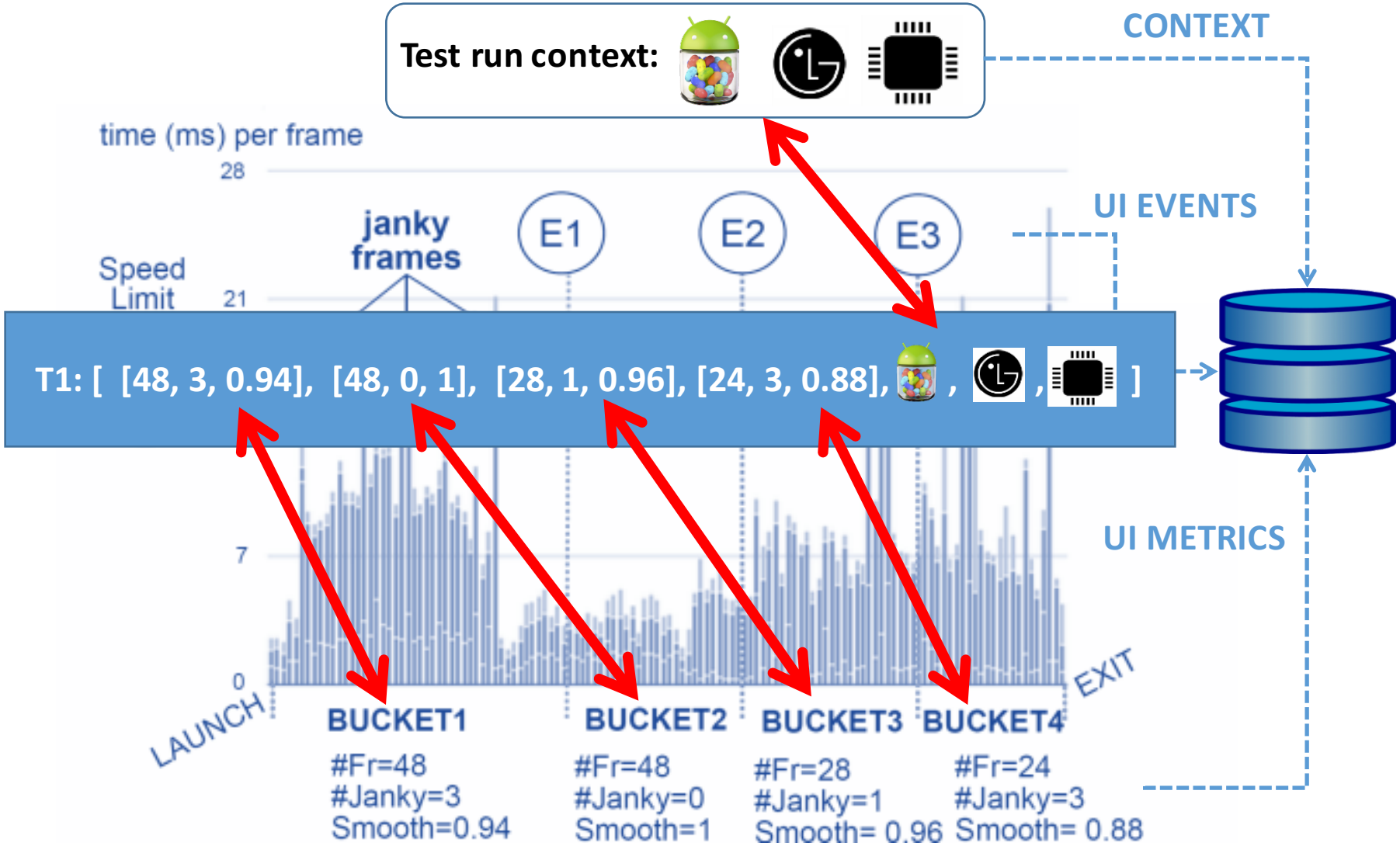
1. Run UI Performance Test

Example. UI metrics collected during a test run



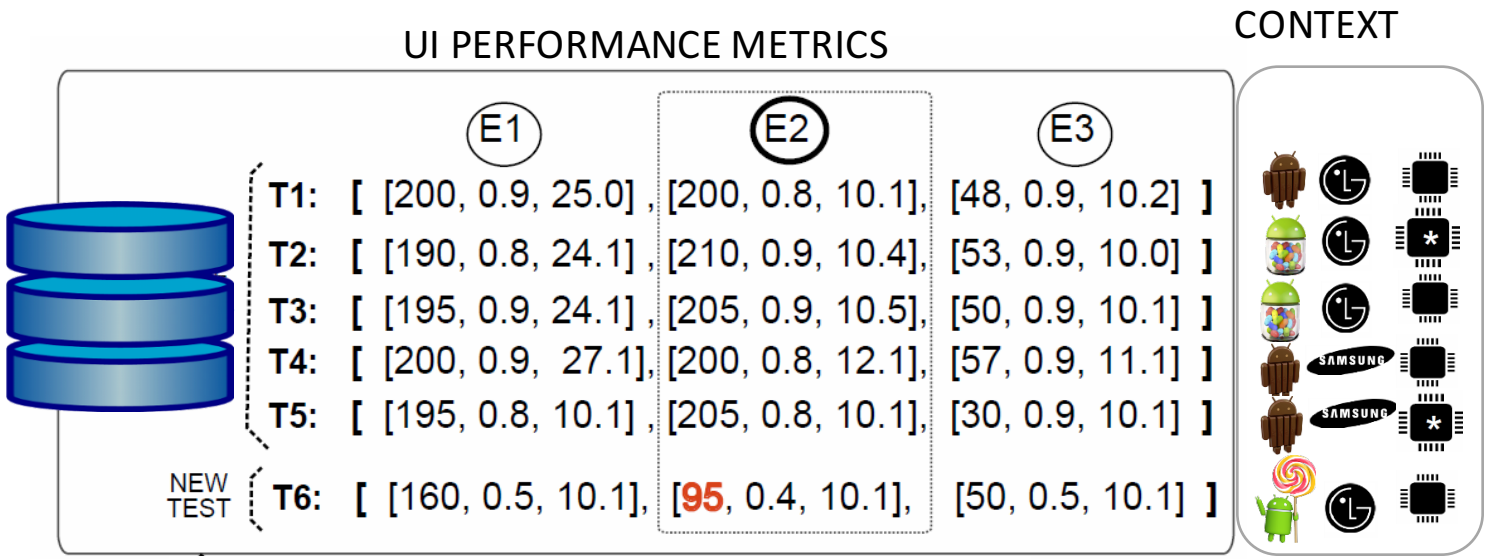
1. Run UI Performance Test

Example. UI metrics collected during a test run



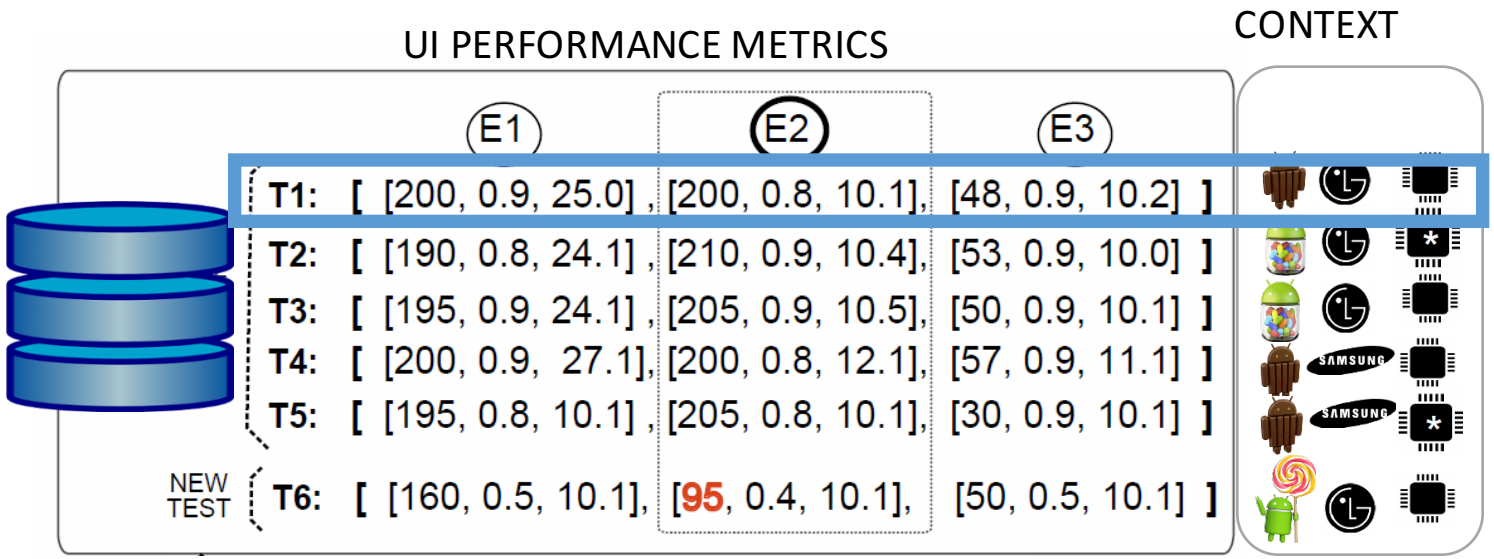
Detecting UI Performance Deviations

2. Building Model of Versions/Contexts



Detecting UI Performance Deviations

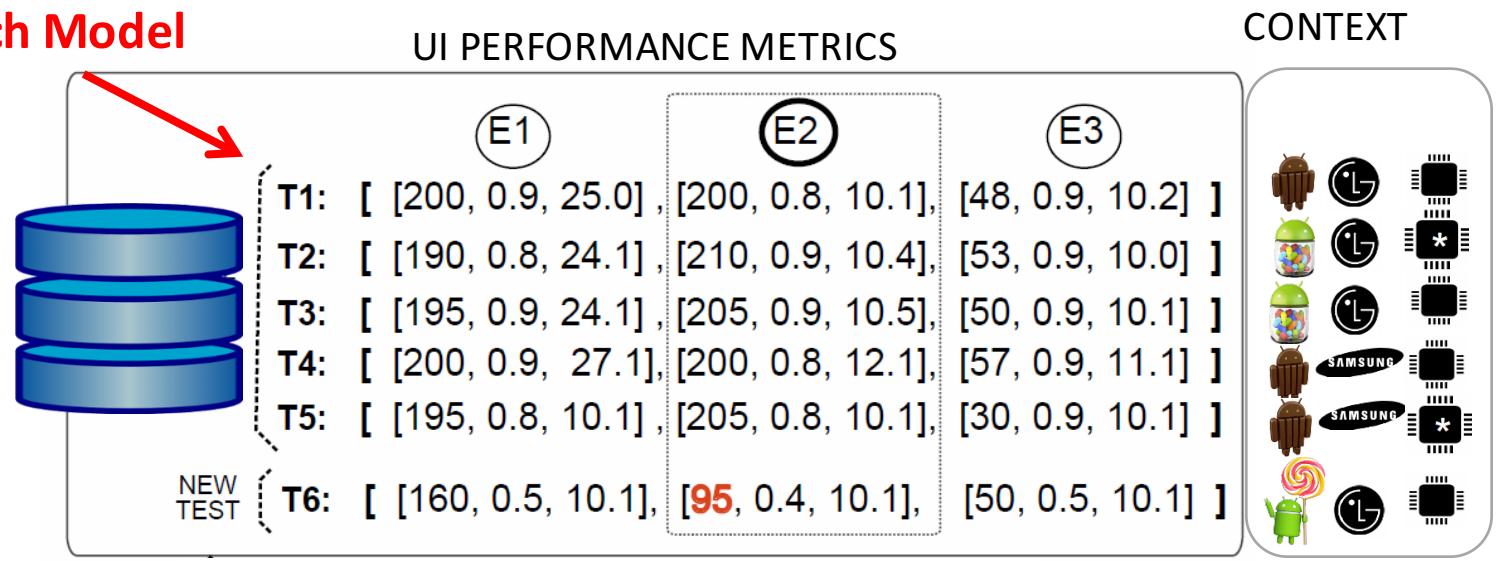
2. Building Model of Versions/Contexts



Detecting UI Performance Deviations

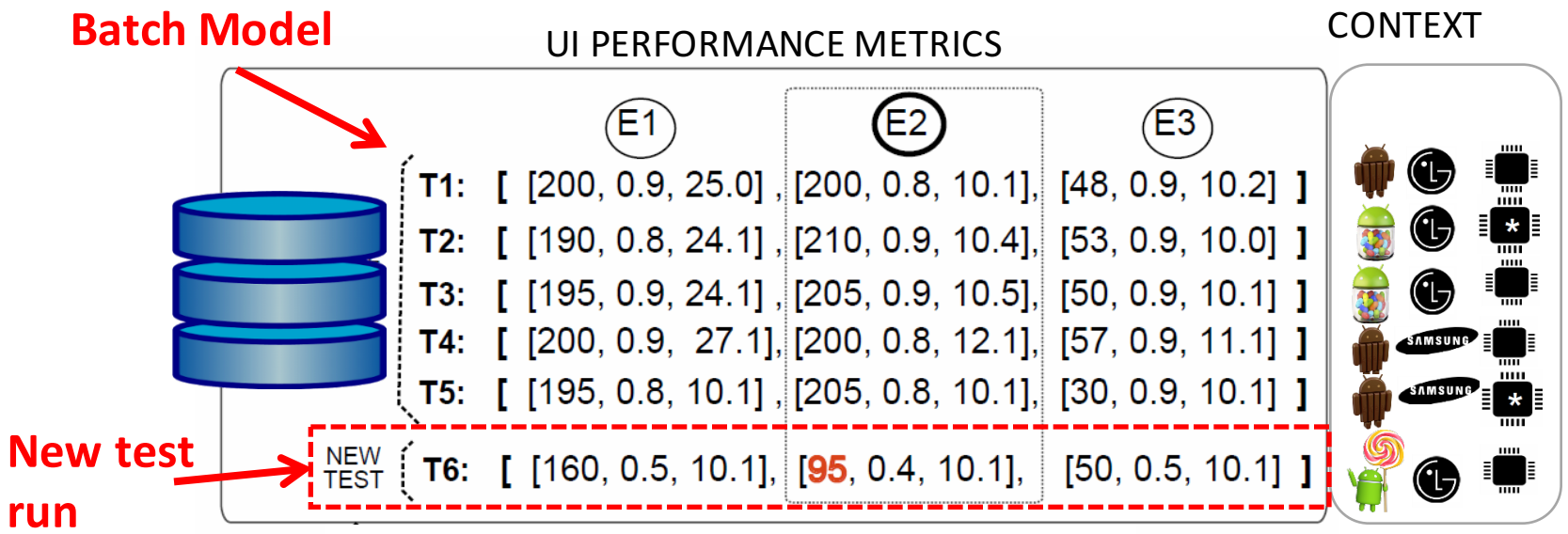
2. Building Model of Versions/Contexts

Batch Model



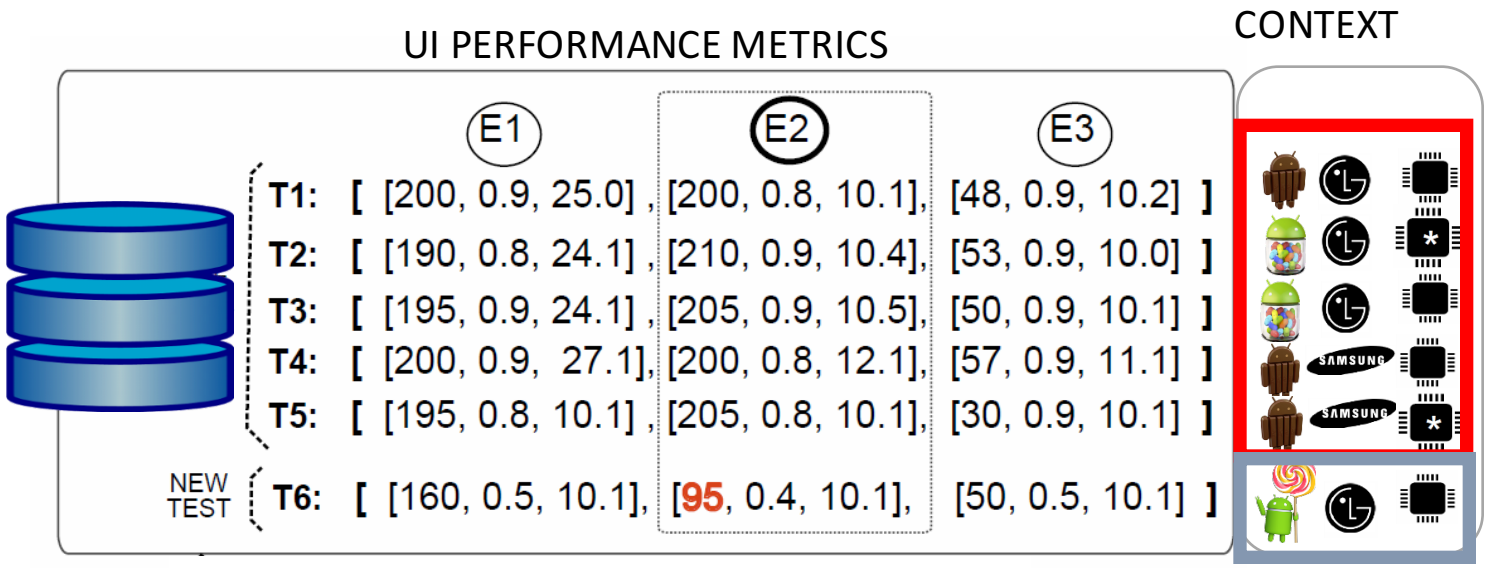
Detecting UI Performance Deviations

2. Building Model of Versions/Contexts



Detecting UI Performance Deviations

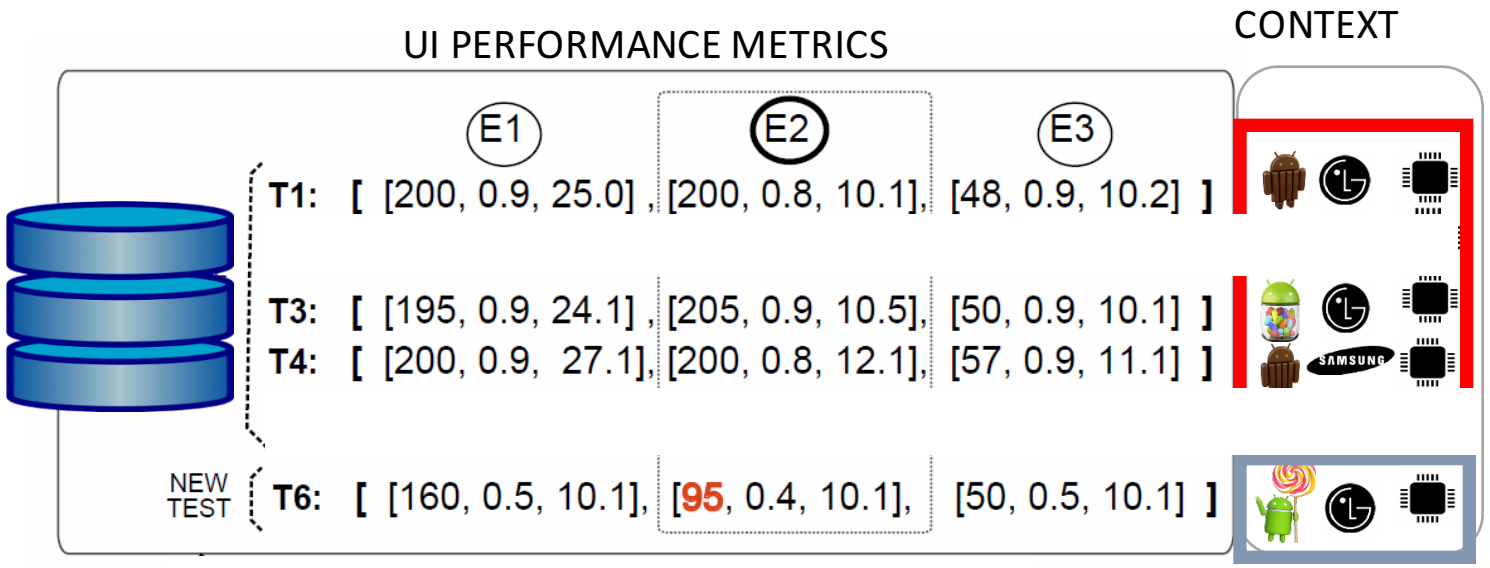
3. Similarity Measure



Rank previous tests with most similar context

Detecting UI Performance Deviations

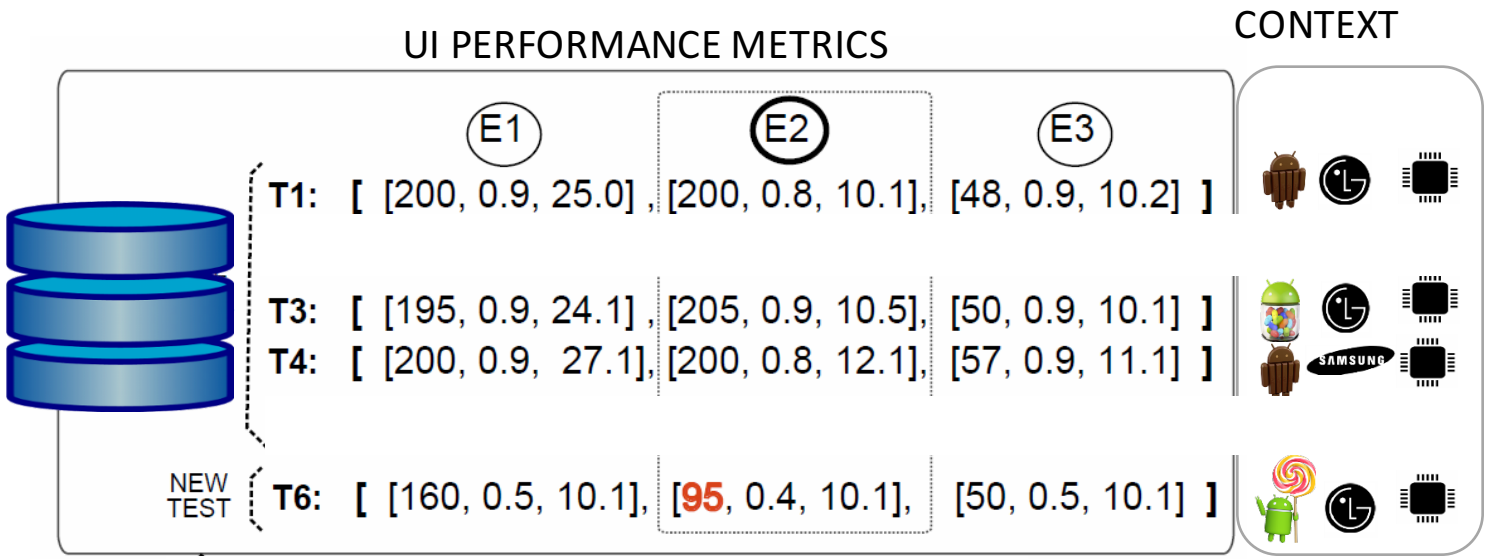
3. Similarity Measure



Rank previous tests with most similar context

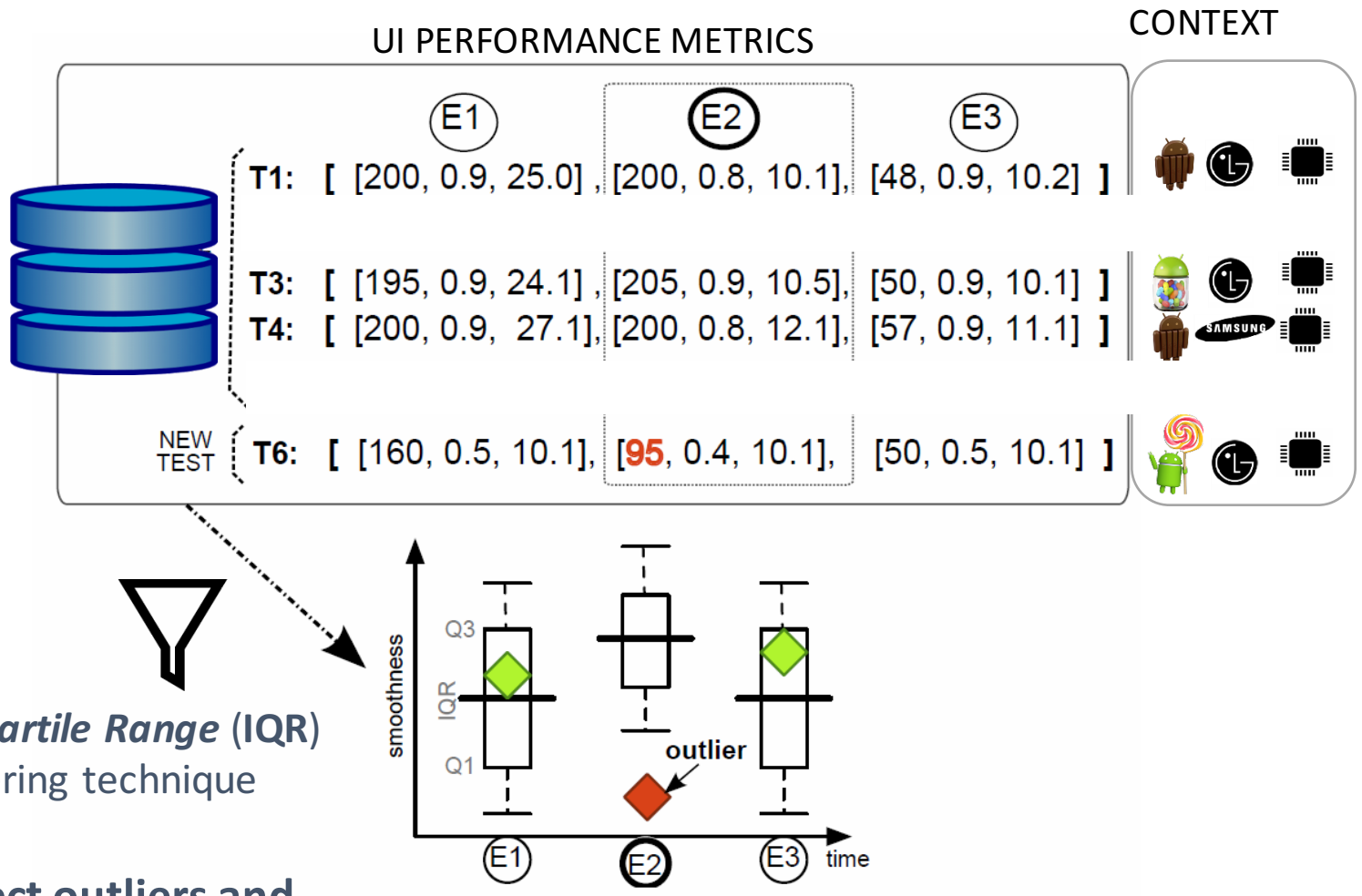
Detecting UI Performance Deviations

4. Performance Outlier Detection



Detecting UI Performance Deviations

4. Performance Outlier Detection

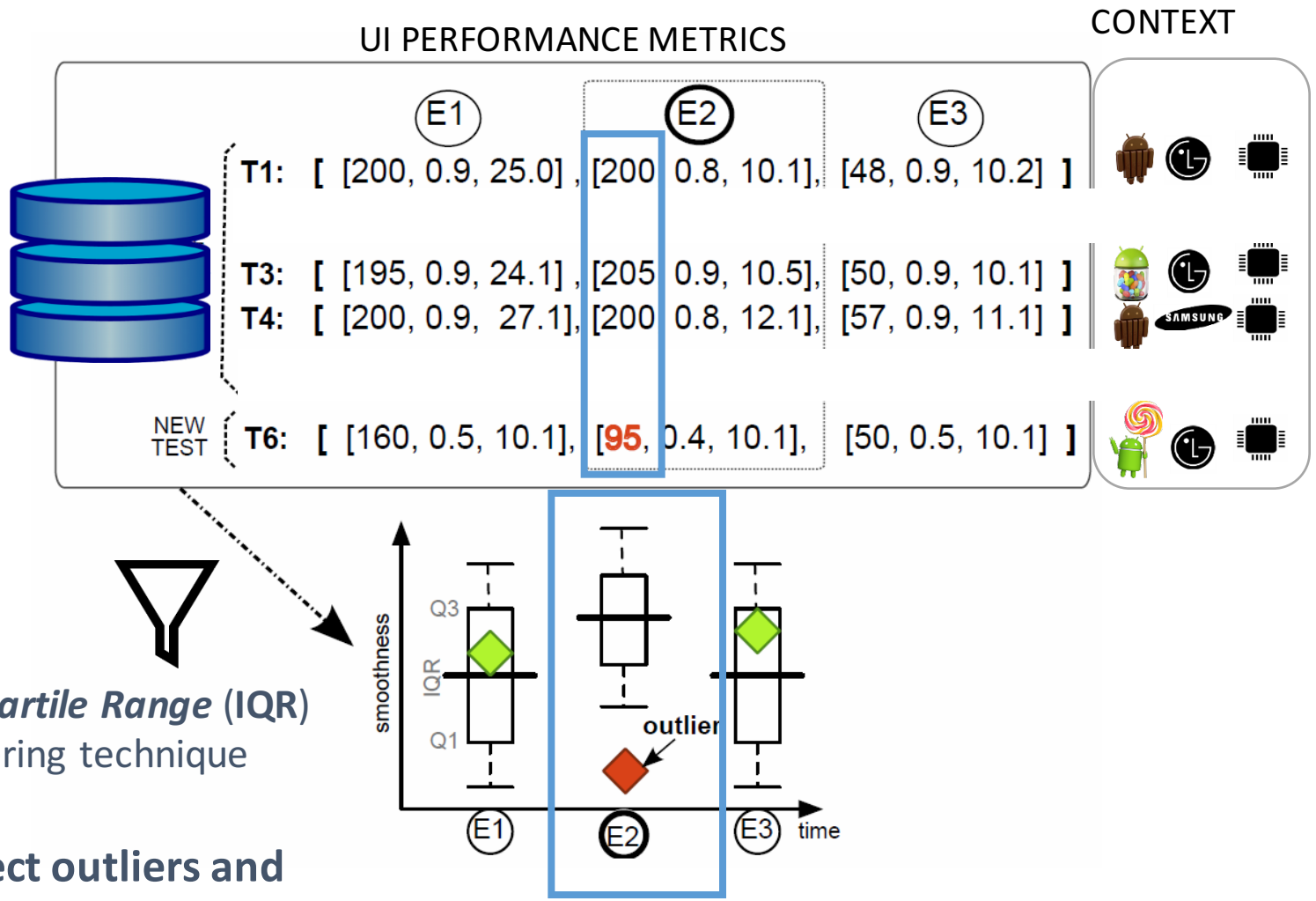


Interquartile Range **(IQR)**
Filtering technique

Detect outliers and extreme values

Detecting UI Performance Deviations

4. Performance Outlier Detection

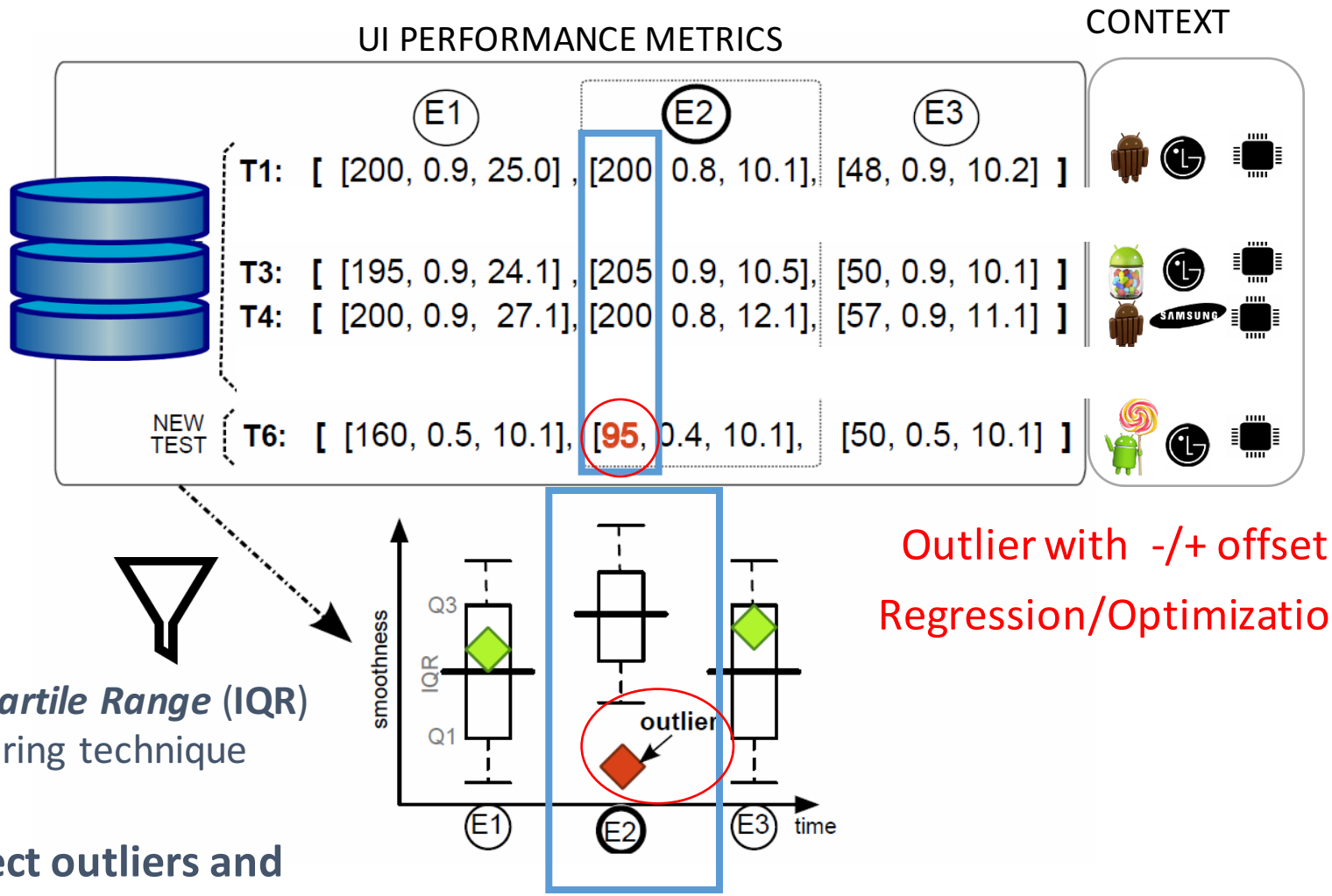


Interquartile Range **(IQR)**
Filtering technique

Detect outliers and extreme values

Detecting UI Performance Deviations

4. Performance Outlier Detection



Interquartile Range **(IQR)**
Filtering technique

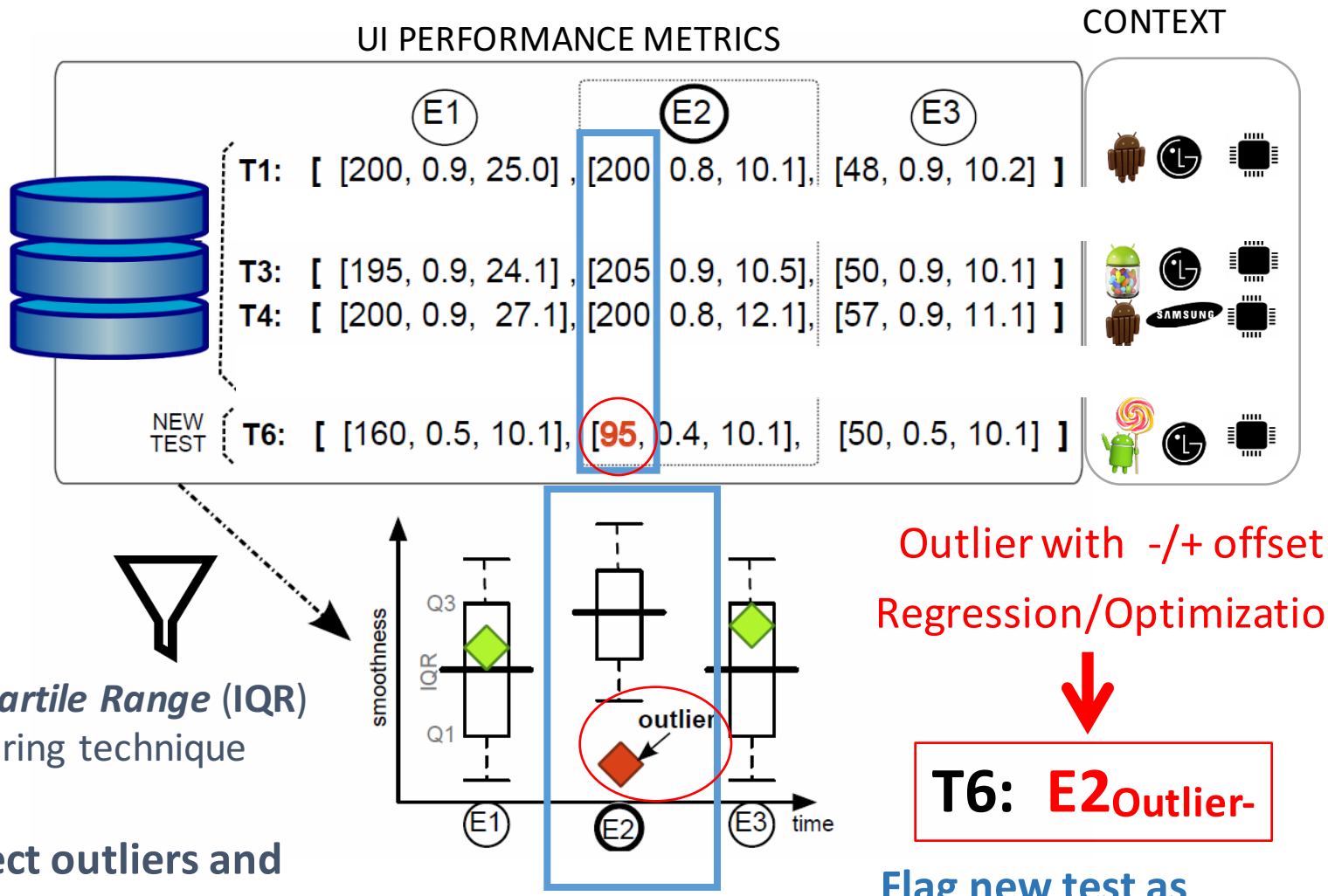
Detect outliers and extreme values

Outlier with -/+ offset

Regression/Optimization

Detecting UI Performance Deviations

4. Performance Outlier Detection



Interquartile Range **(IQR)**
Filtering technique

Detect outliers and extreme values

Outlier with -/+ offset

Regression/Optimization

T6: E2Outlier-

Flag new test as regression in event E2!

Detecting UI Performance Deviations

4. Identify Context Patterns

- Identify common ***context patterns*** that induce outliers
  - Help developers to isolate defective contexts

- ***Association Rule Mining*** *(Apriori algorithm)*

Examples of rules:

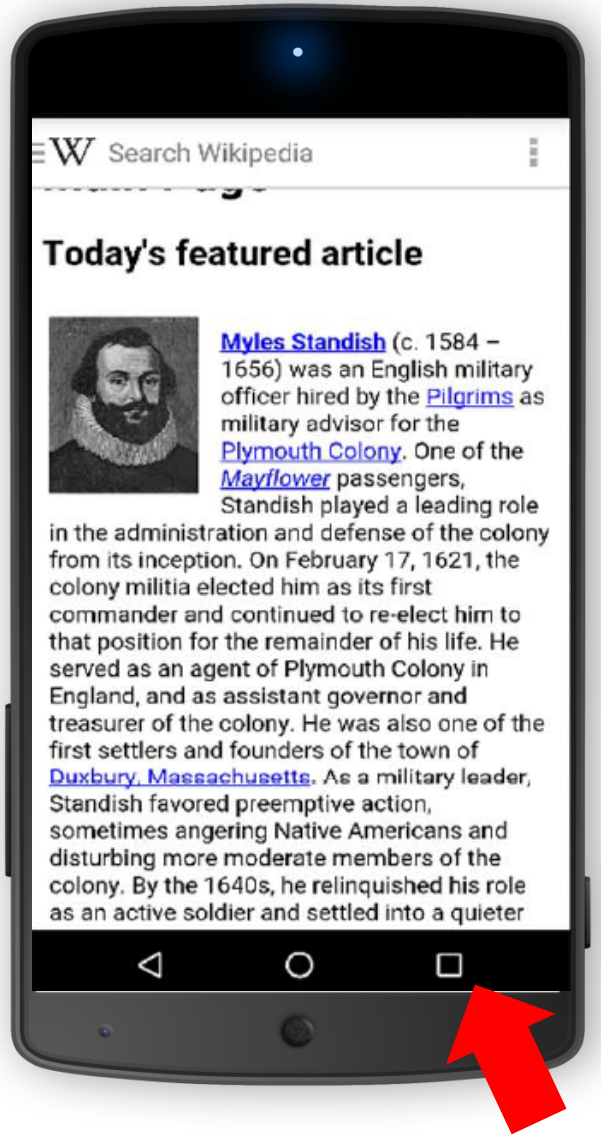
{sdk=5.1} => Outlier+
{v=1.2, dev=LG} => E2Outlier-

Leveraging Crowds of Users to Reproduce App Crashes

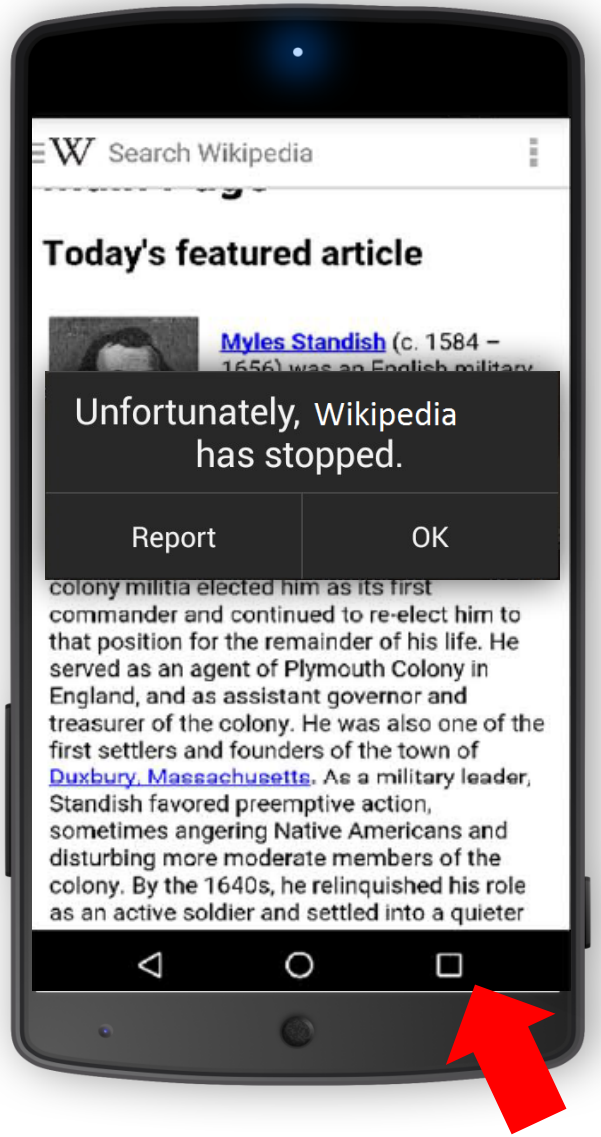
María Gómez,
Bram Adams (Poly. Montréal),
Walid Maalej (Univ. Hamburg)

Problem: How to reproduce the crash?

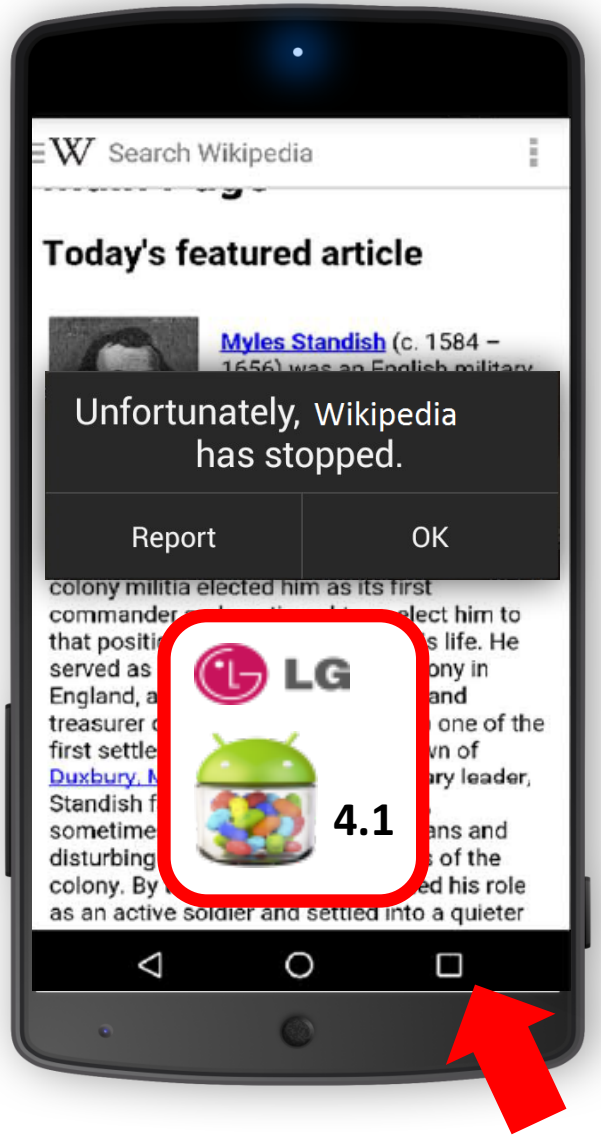
Problem: How to reproduce the crash?



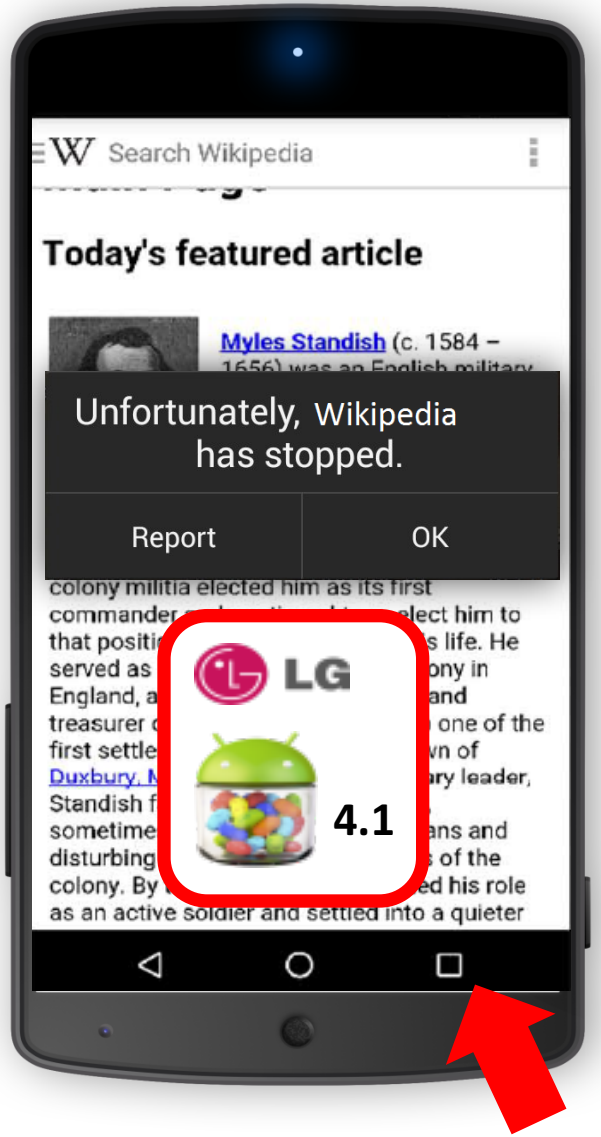
Problem: How to reproduce the crash?



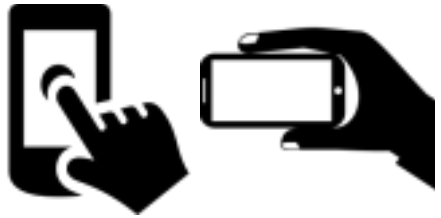
Problem: How to reproduce the crash?



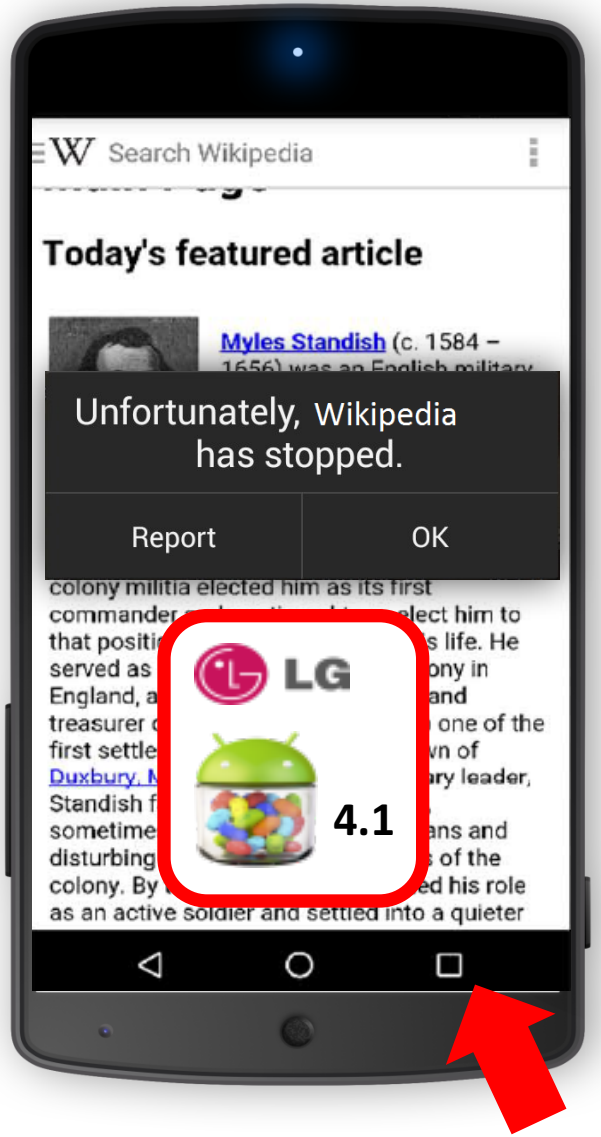
Problem: How to reproduce the crash?



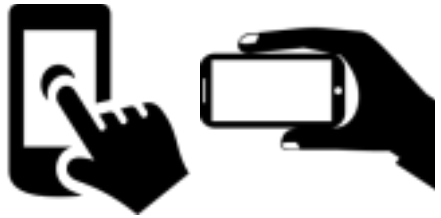
User Interactions



Problem: How to reproduce the crash?



User Interactions



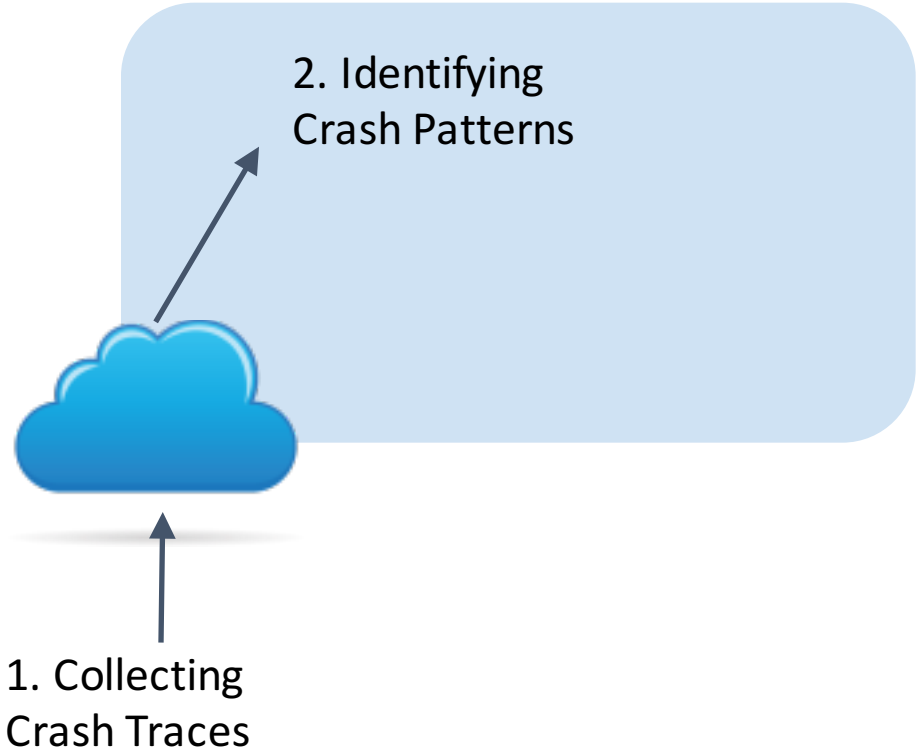
Execution Context (static/dynamic)

MoTiF: Crash Reproduction

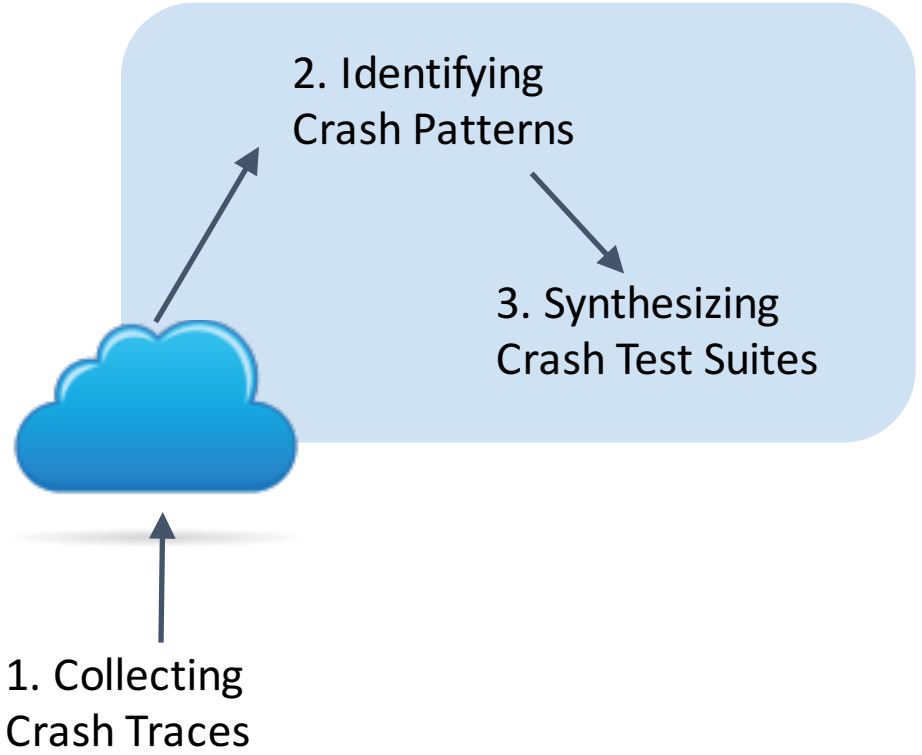
MoTiF: Crash Reproduction

1. Collecting
Crash Traces

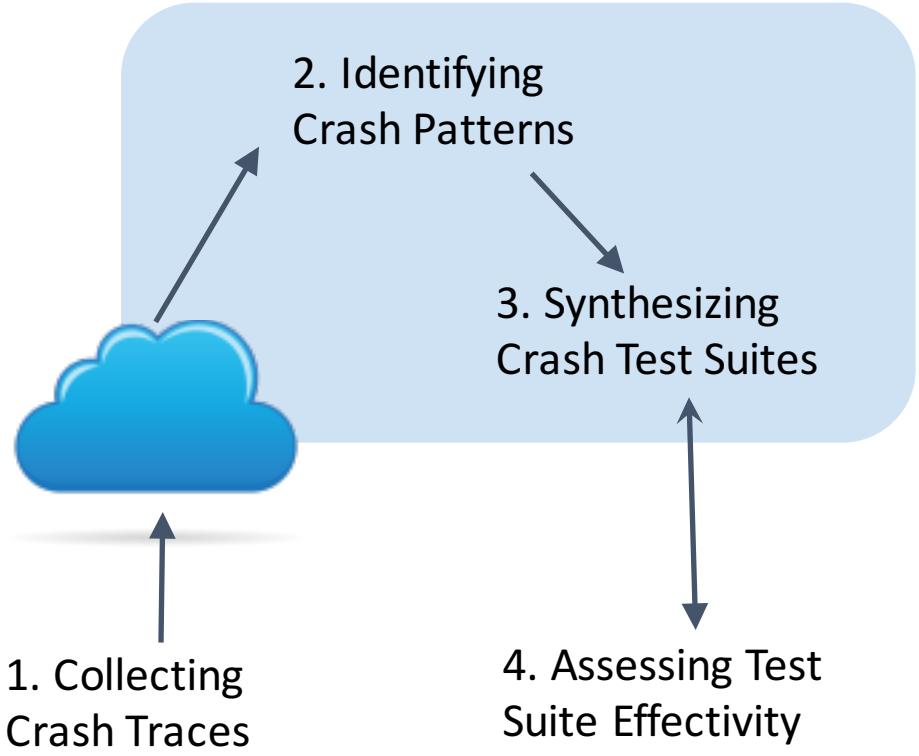
MoTiF: Crash Reproduction



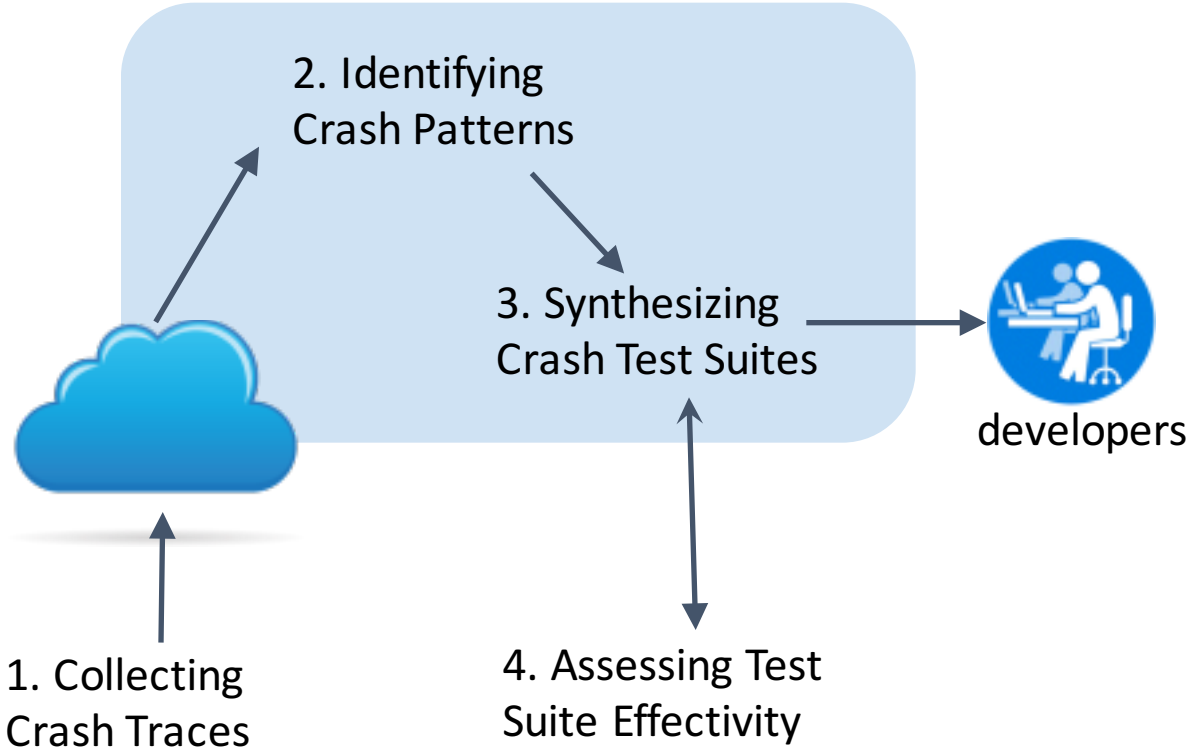
MoTiF: Crash Reproduction



MoTiF: Crash Reproduction

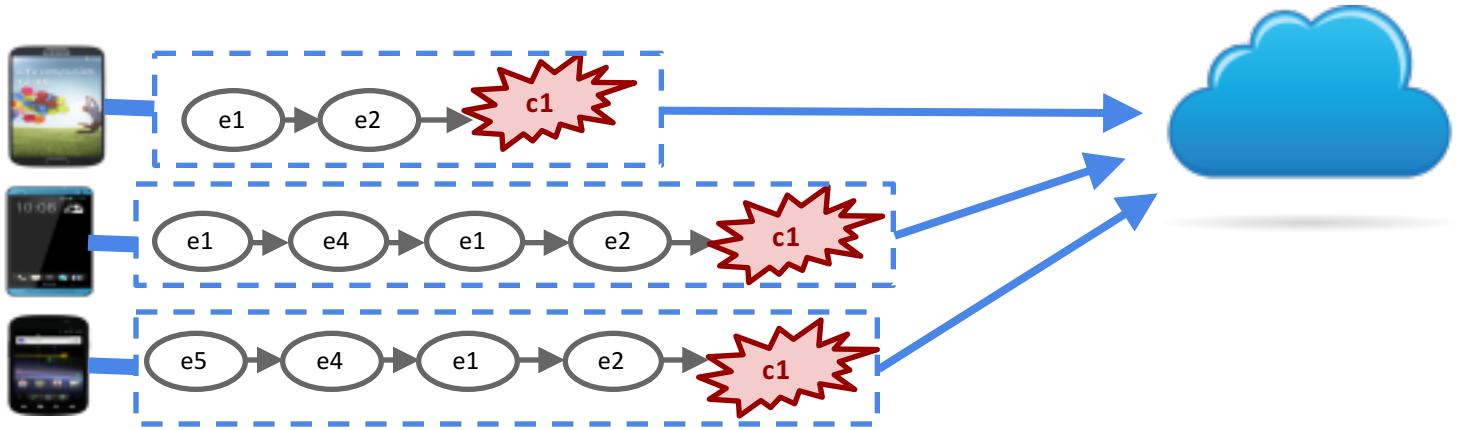


MoTiF: Crash Reproduction



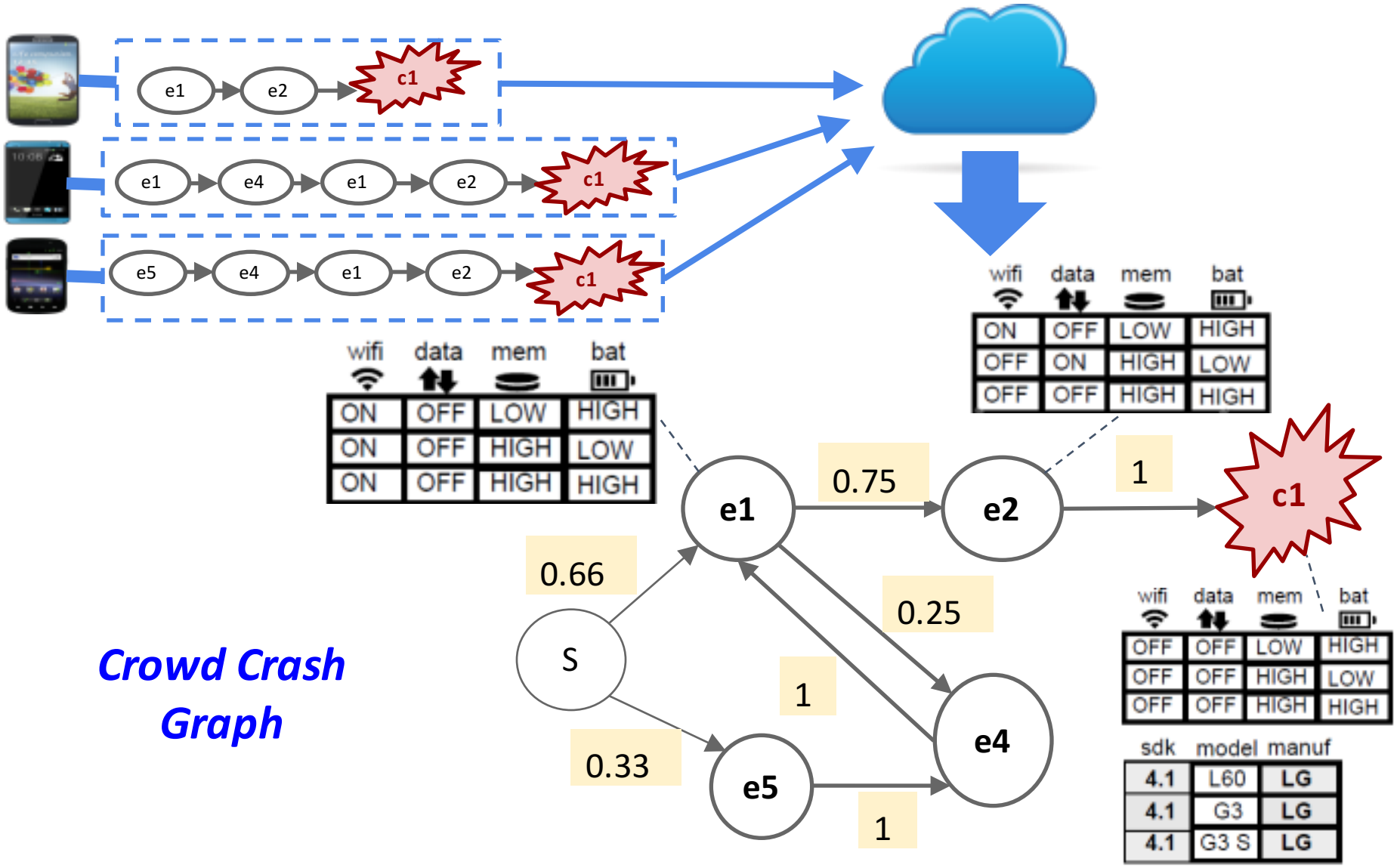
2. Identifying Crash Patterns

Crowd Crash Graph



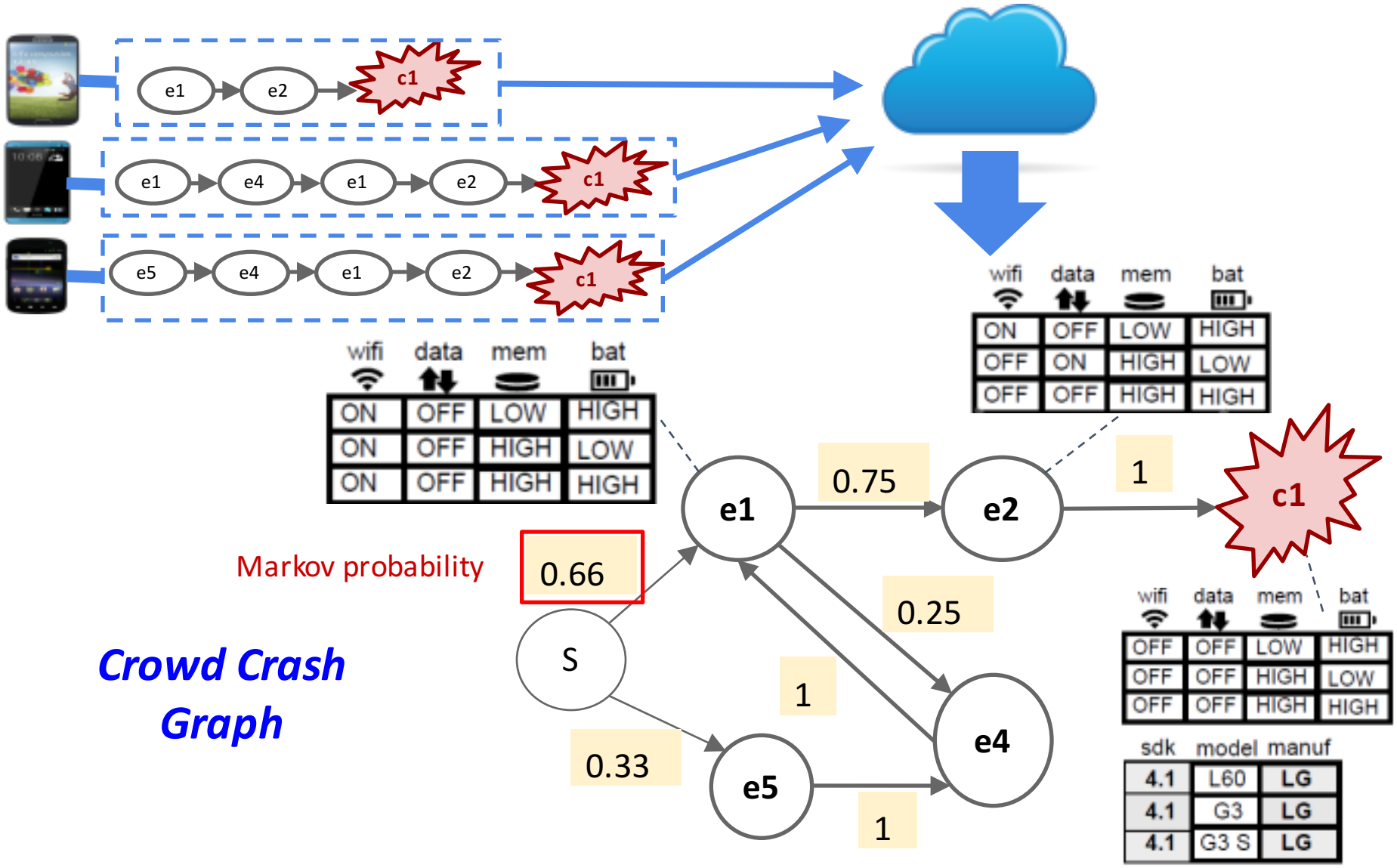
2. Identifying Crash Patterns

Crowd Crash Graph



2. Identifying Crash Patterns

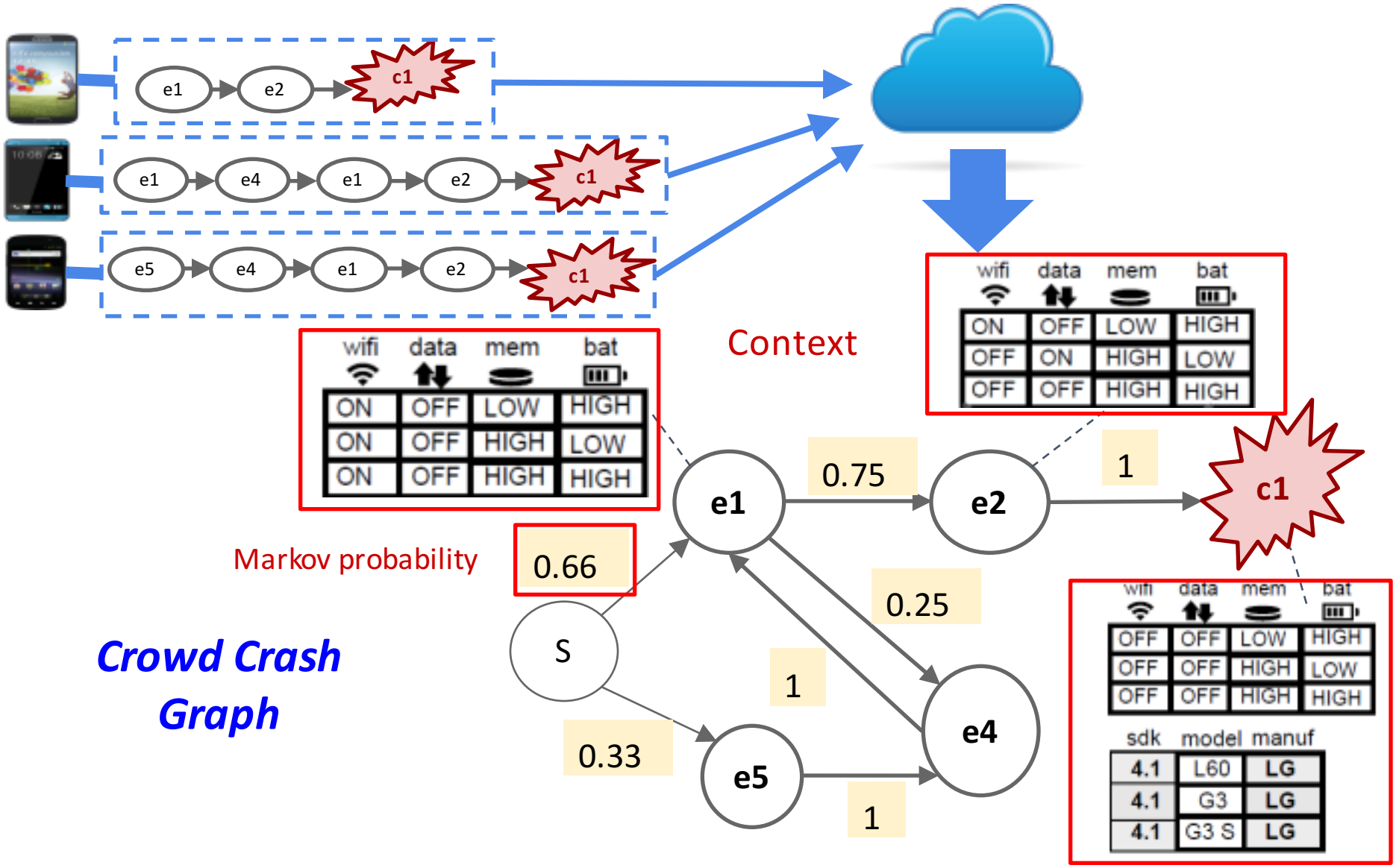
Crowd Crash Graph



Crowd Crash Graph

2. Identifying Crash Patterns

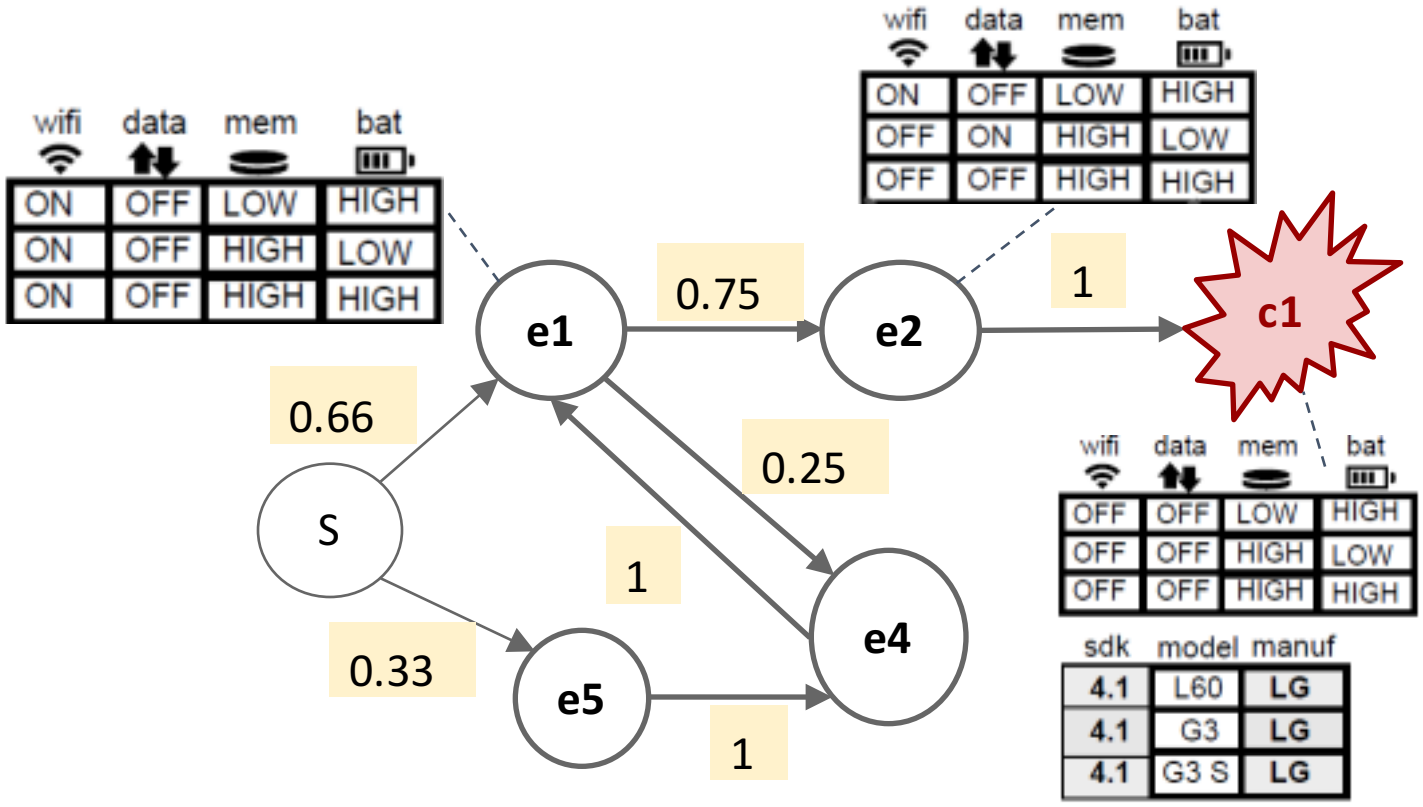
Crowd Crash Graph



2. Identifying Crash Patterns

Crowd Crash Graph

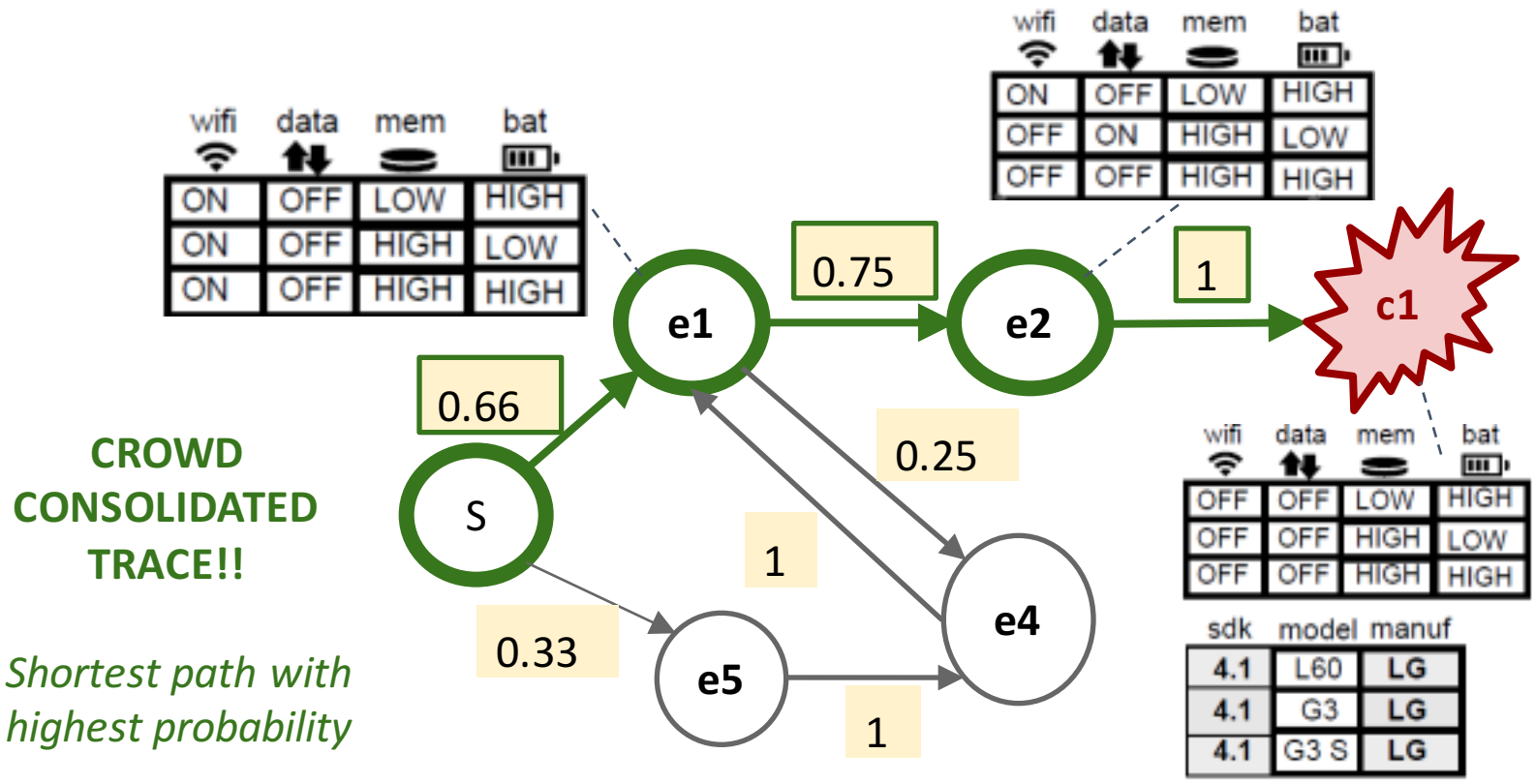
Synthesize steps to reproduce crashes



2. Identifying Crash Patterns

Crowd Crash Graph

Synthesize steps to reproduce crashes

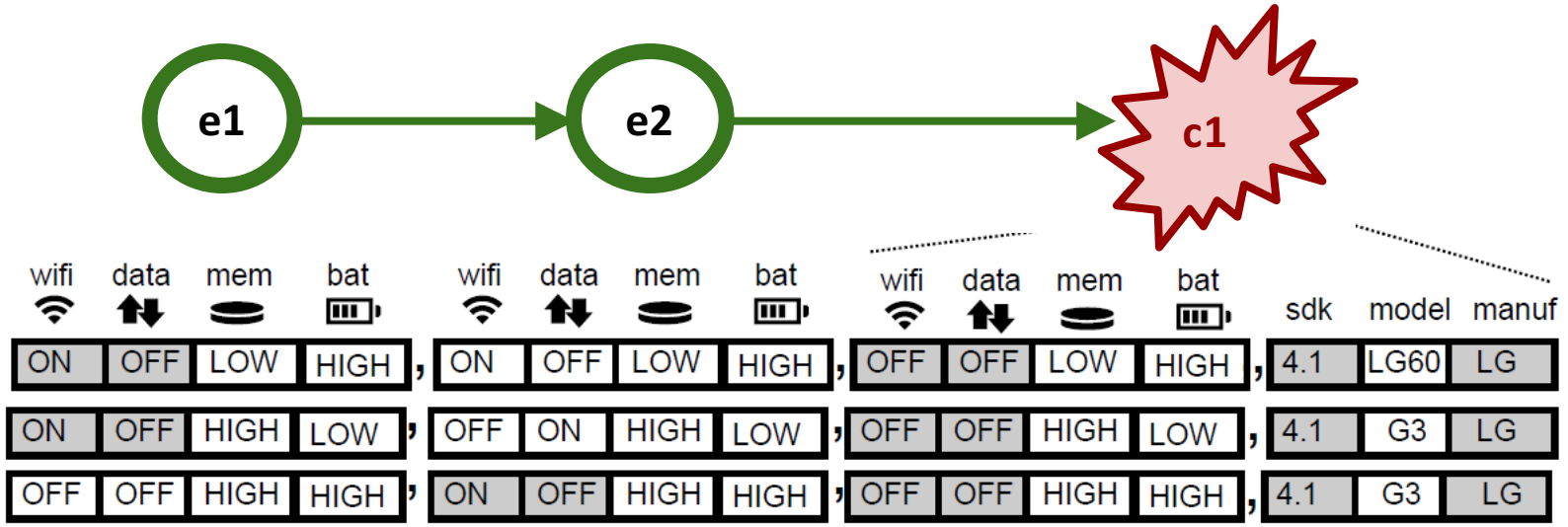


CROWD CONSOLIDATED TRACE!!

Shortest path with highest probability

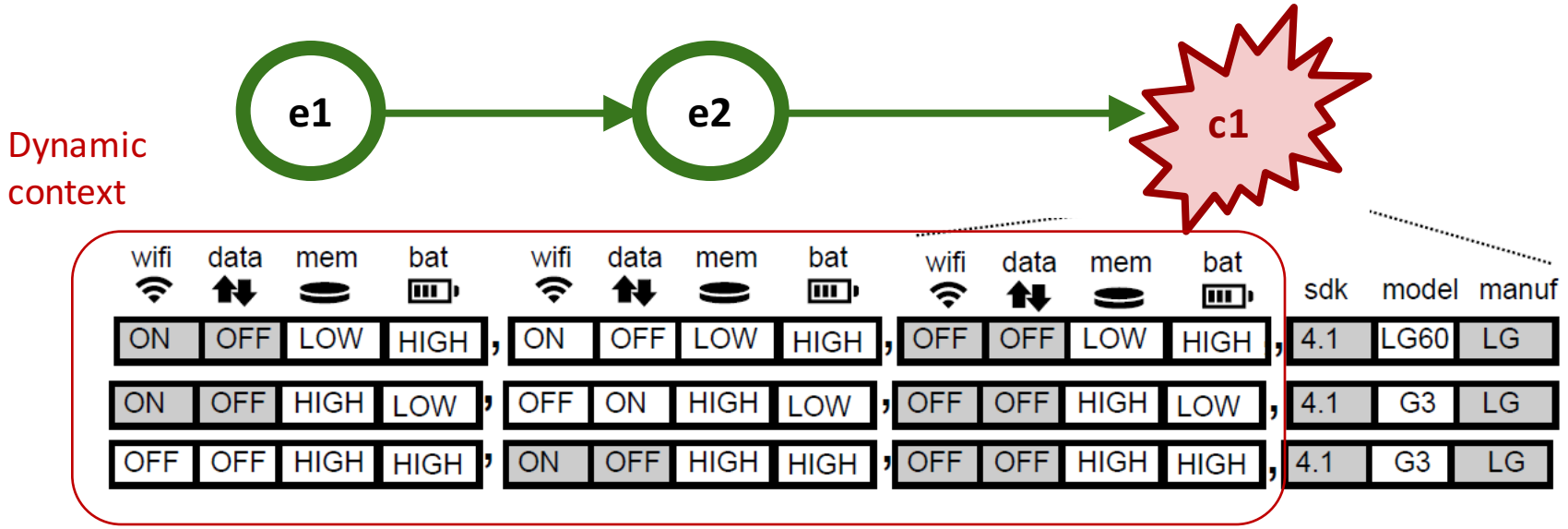
2. Identifying Crash Patterns

Learning crash-prone contexts



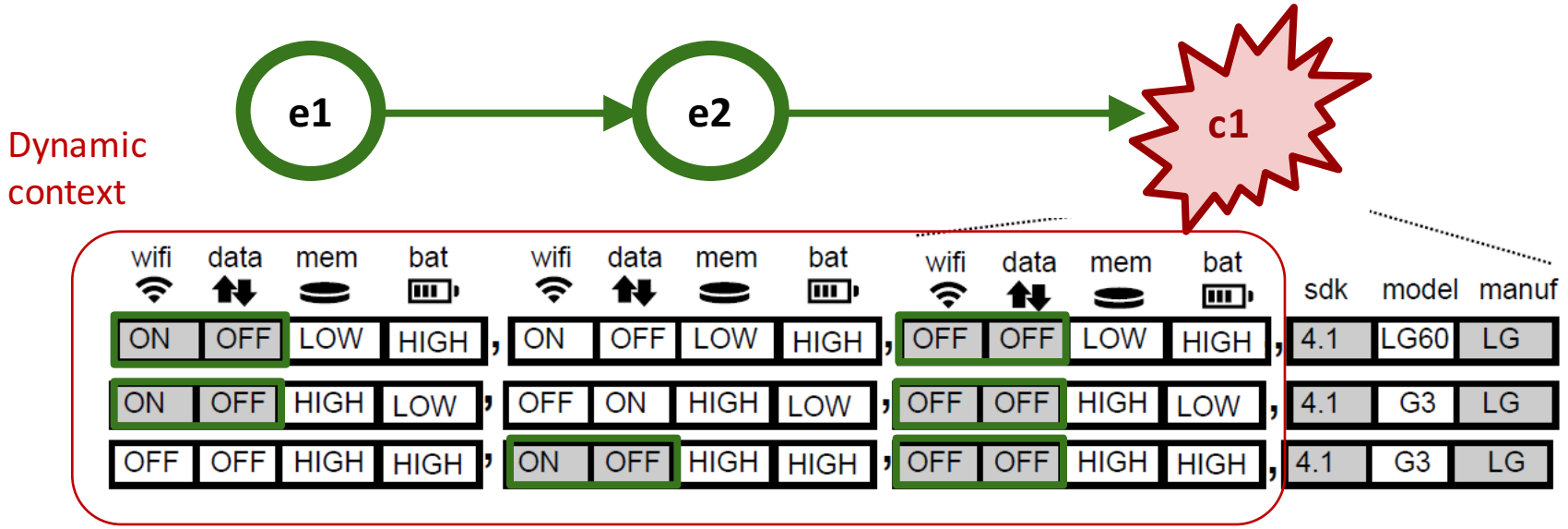
2. Identifying Crash Patterns

Learning crash-prone contexts



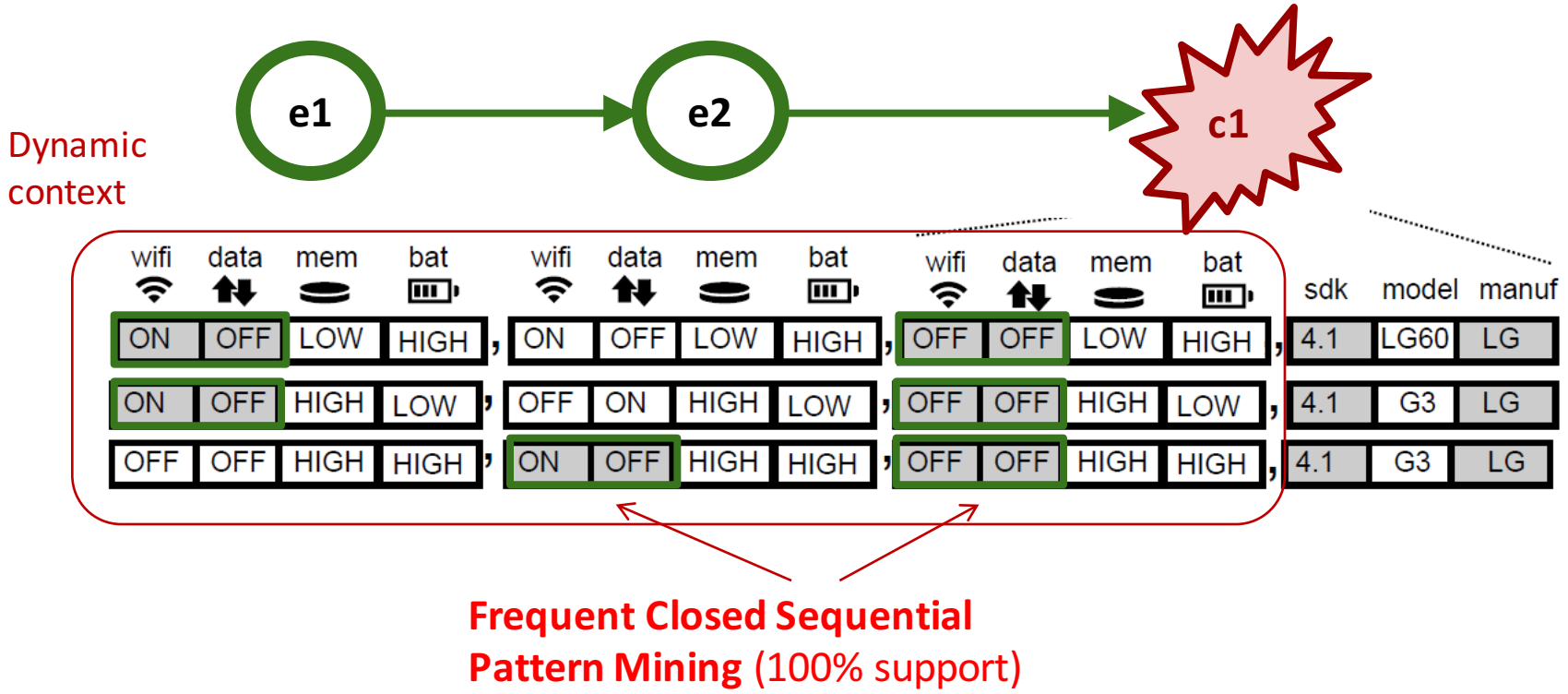
2. Identifying Crash Patterns

Learning crash-prone contexts



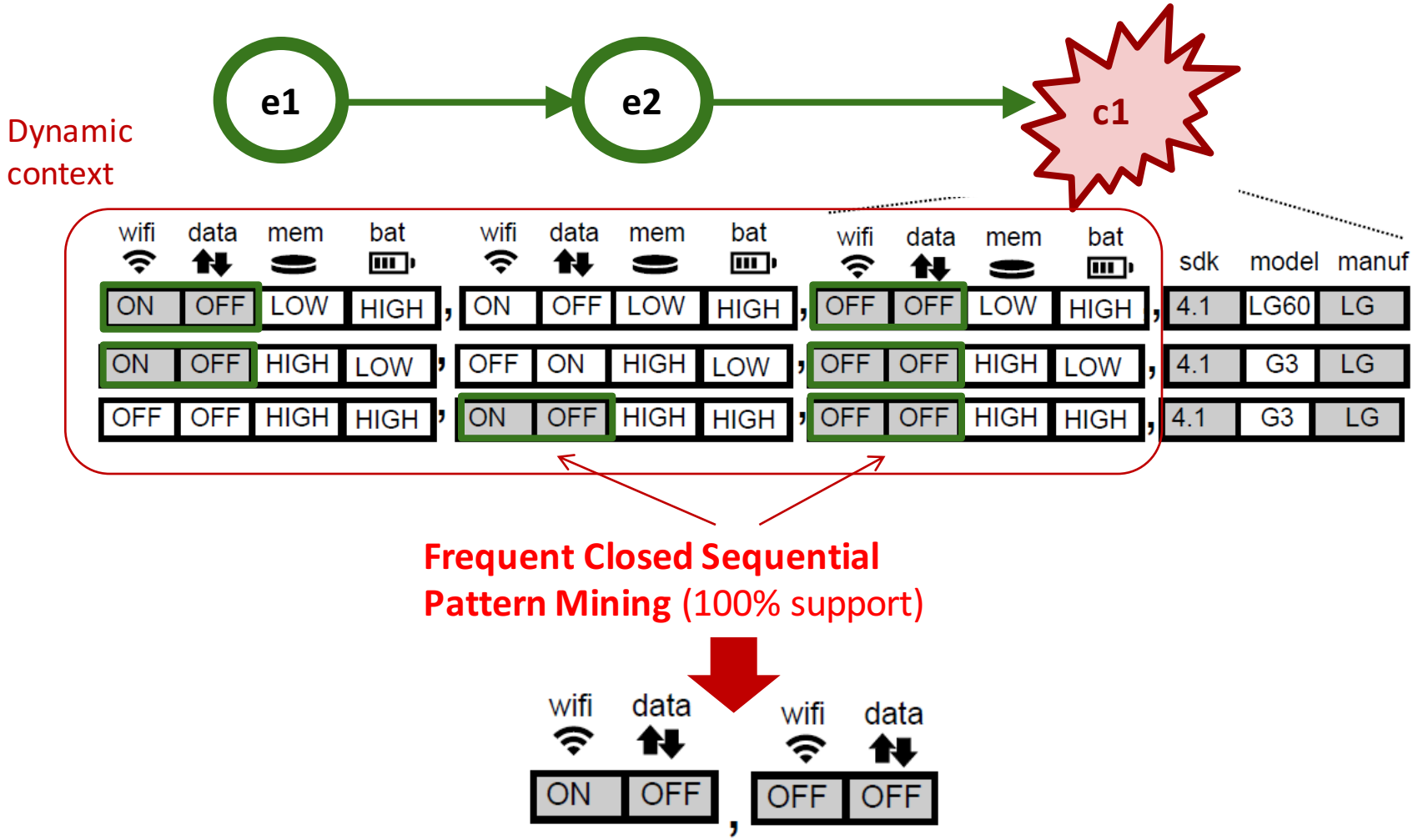
2. Identifying Crash Patterns

Learning crash-prone contexts



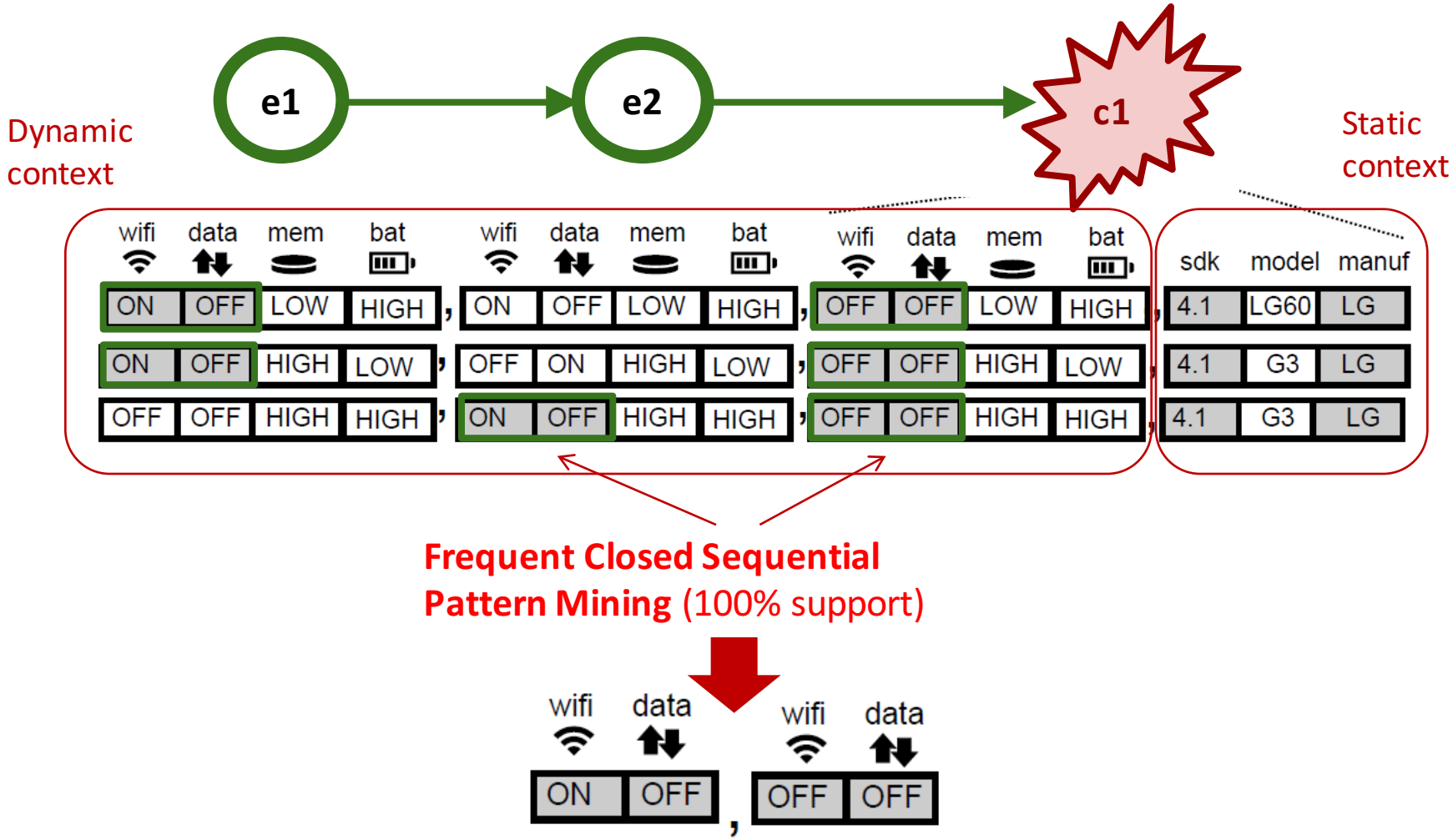
2. Identifying Crash Patterns

Learning crash-prone contexts



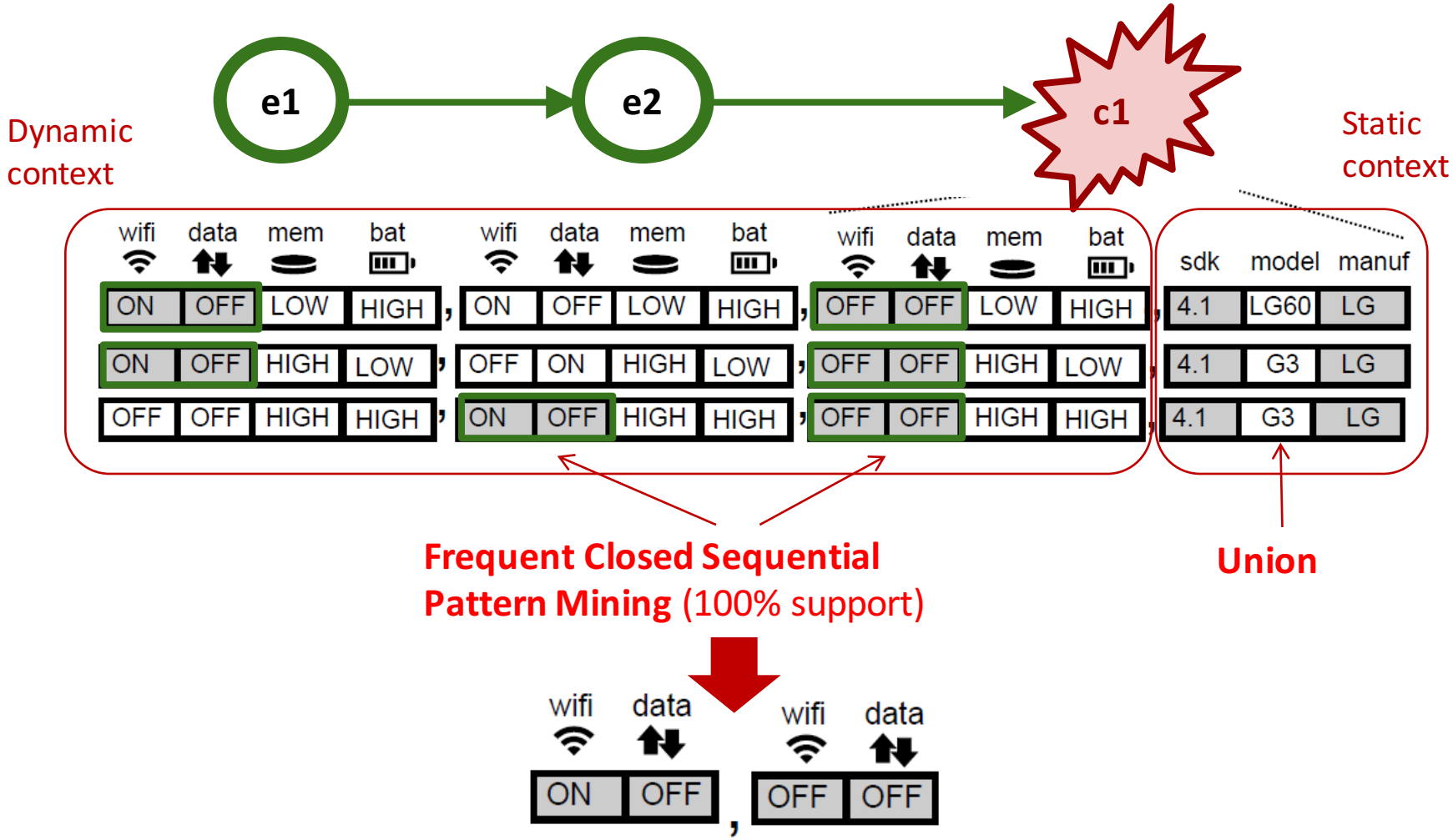
2. Identifying Crash Patterns

Learning crash-prone contexts



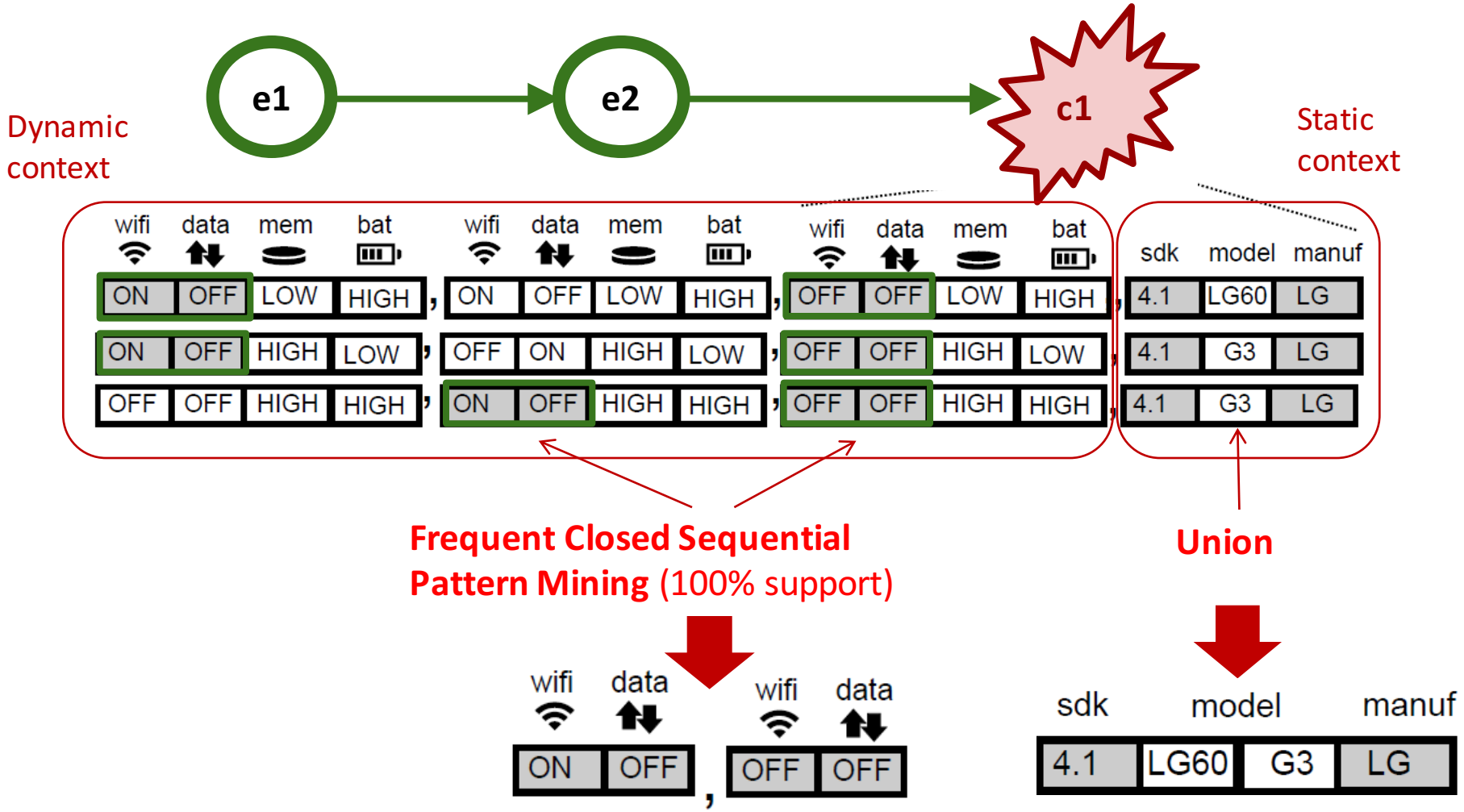
2. Identifying Crash Patterns

Learning crash-prone contexts

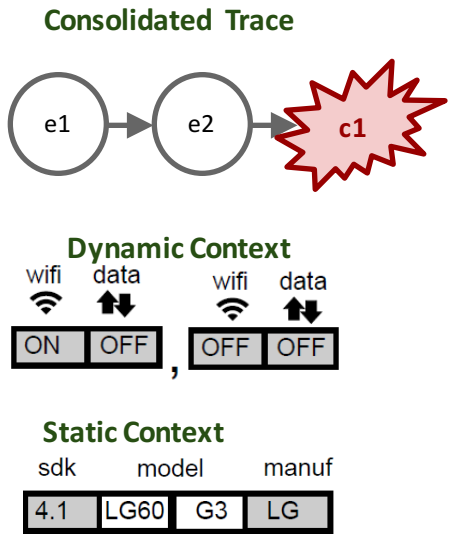


2. Identifying Crash Patterns

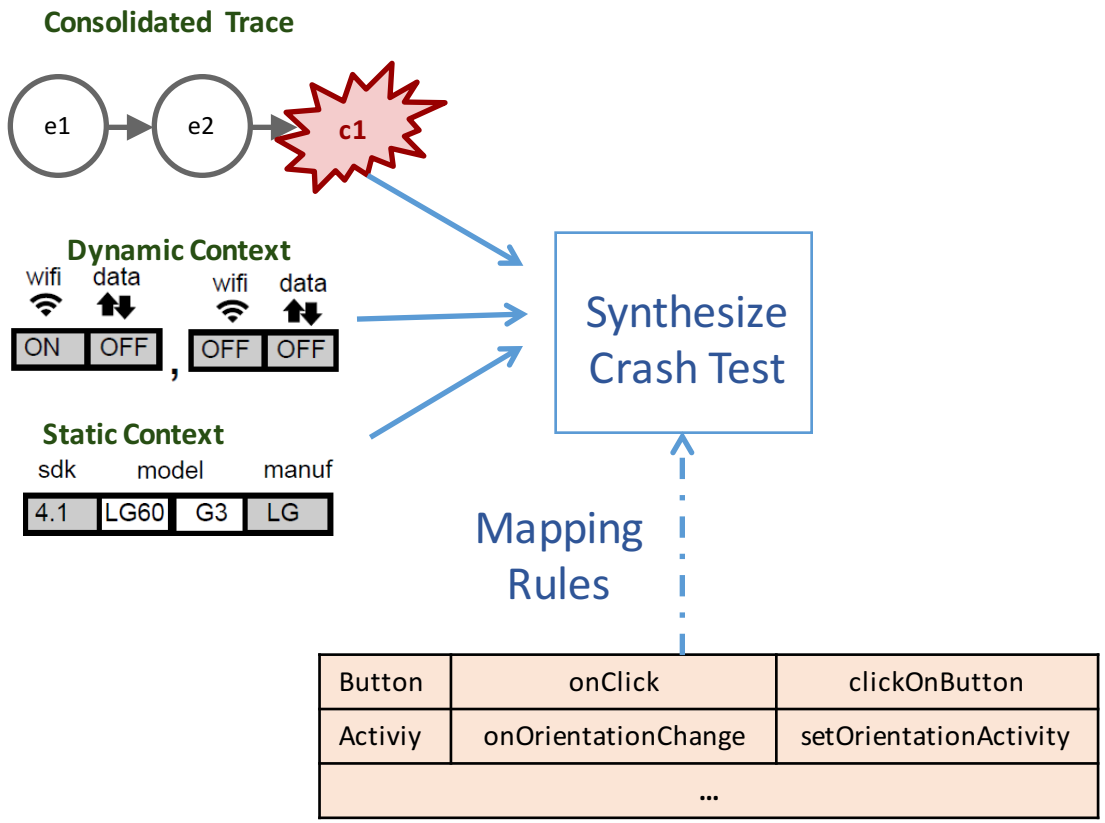
Learning crash-prone contexts



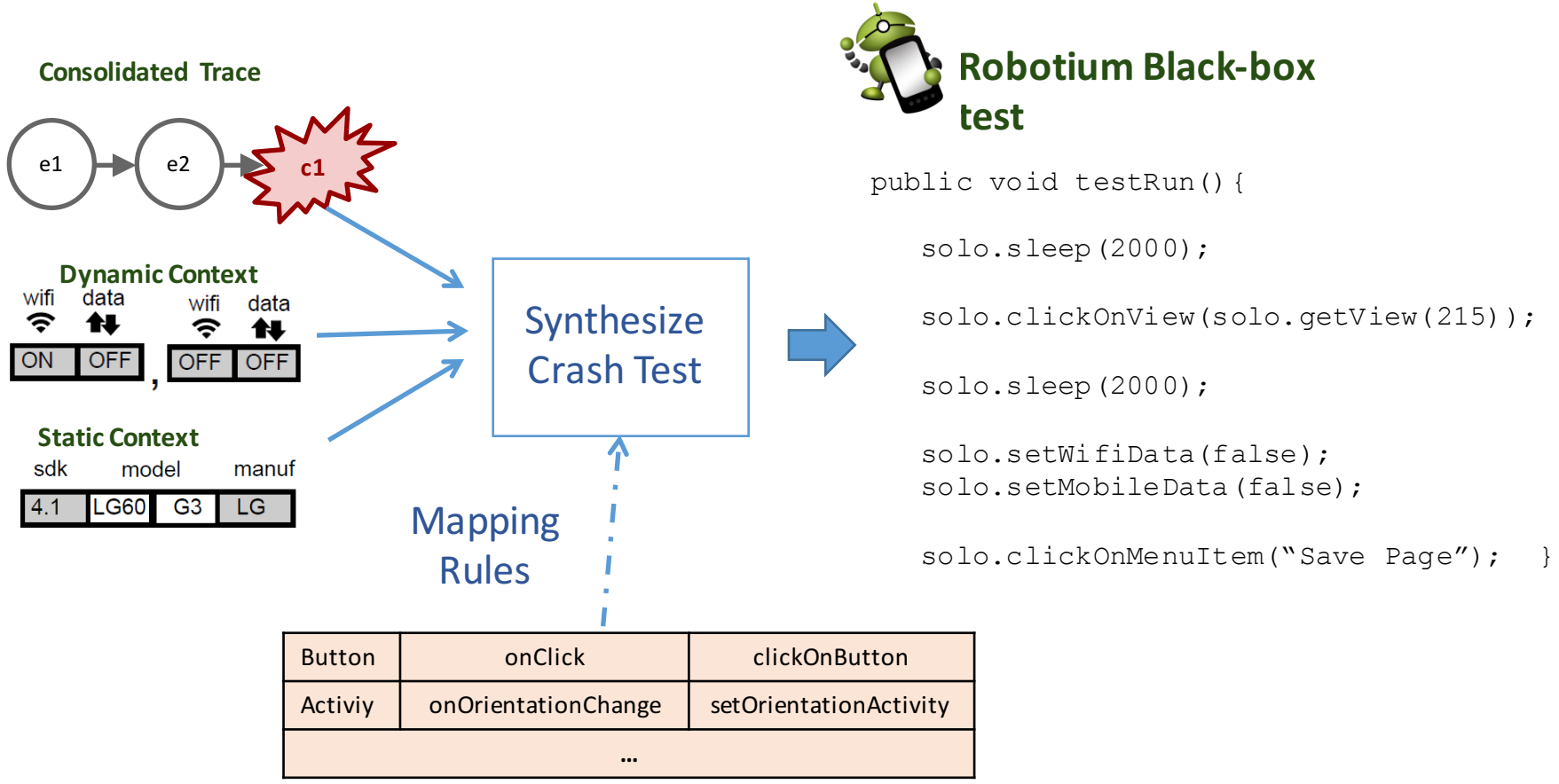
3. Synthesizing Crash Test Suites



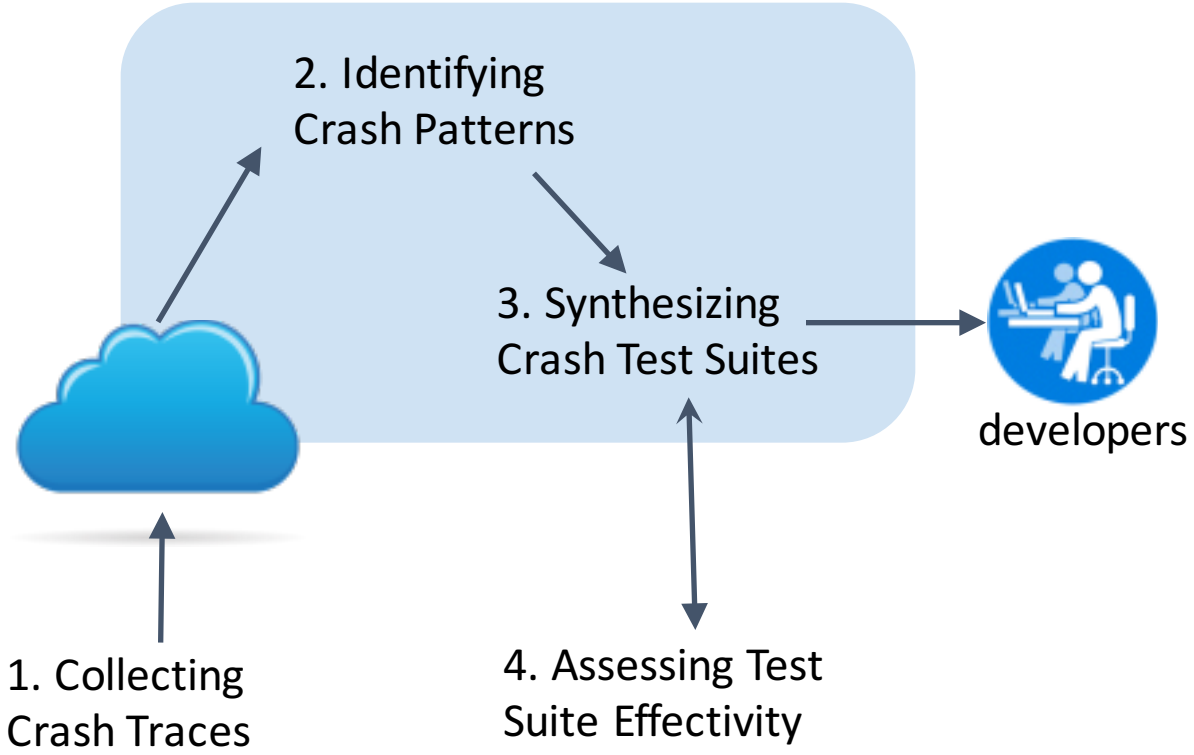
3. Synthesizing Crash Test Suites



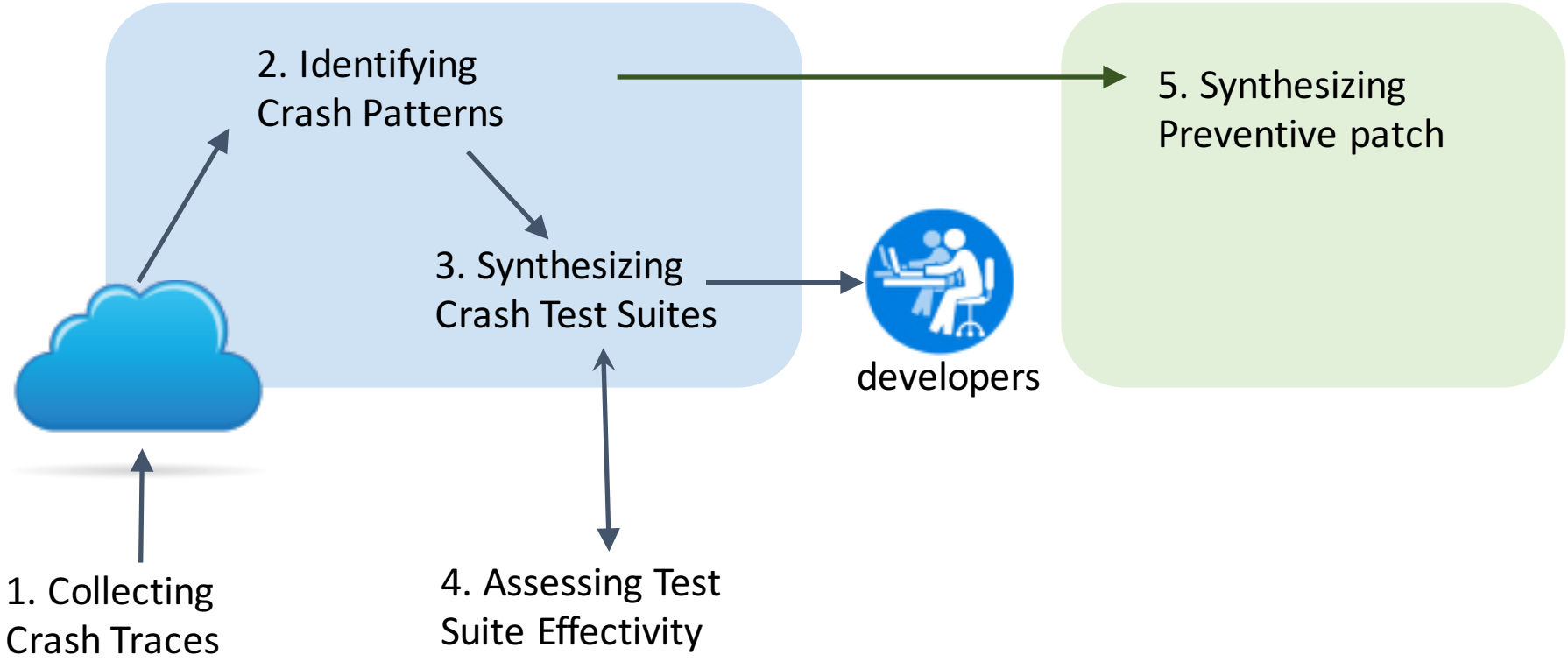
3. Synthesizing Crash Test Suites



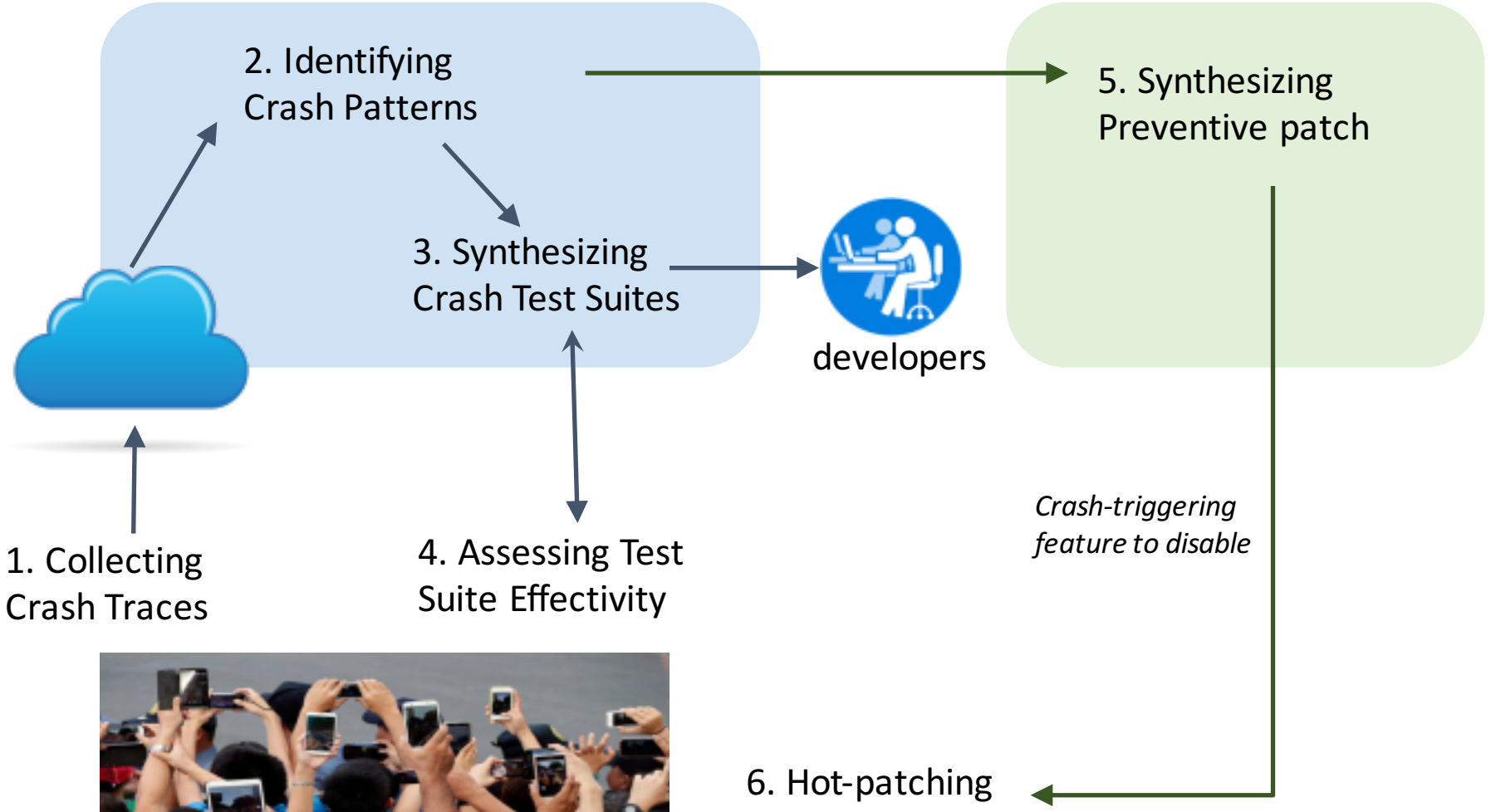
CrowdSeer: Crash Prevention



CrowdSeer: Crash Prevention

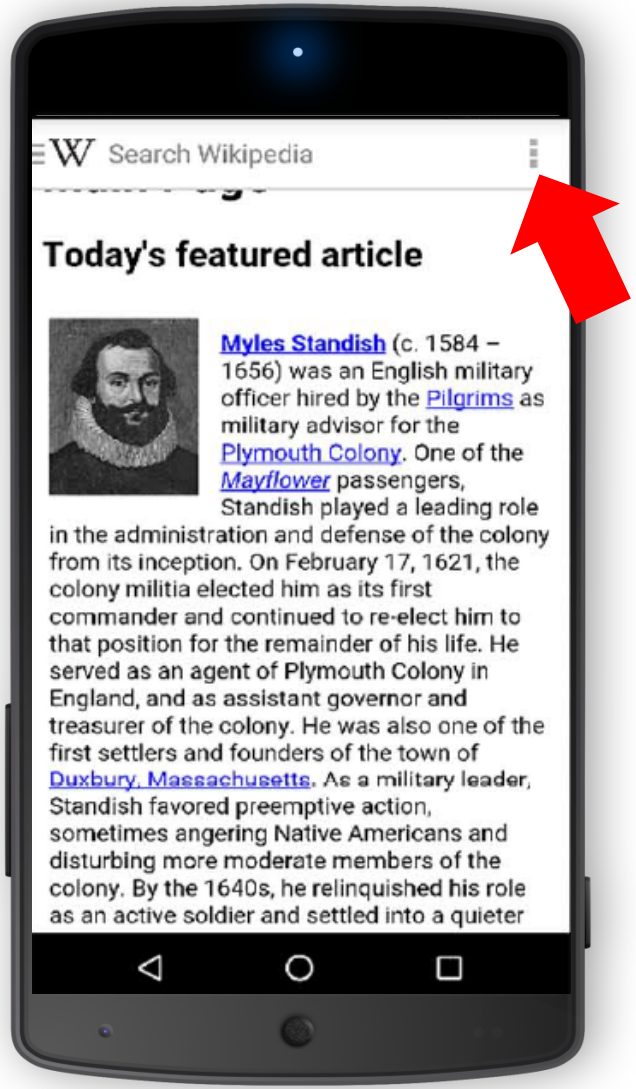


CrowdSeer: Crash Prevention

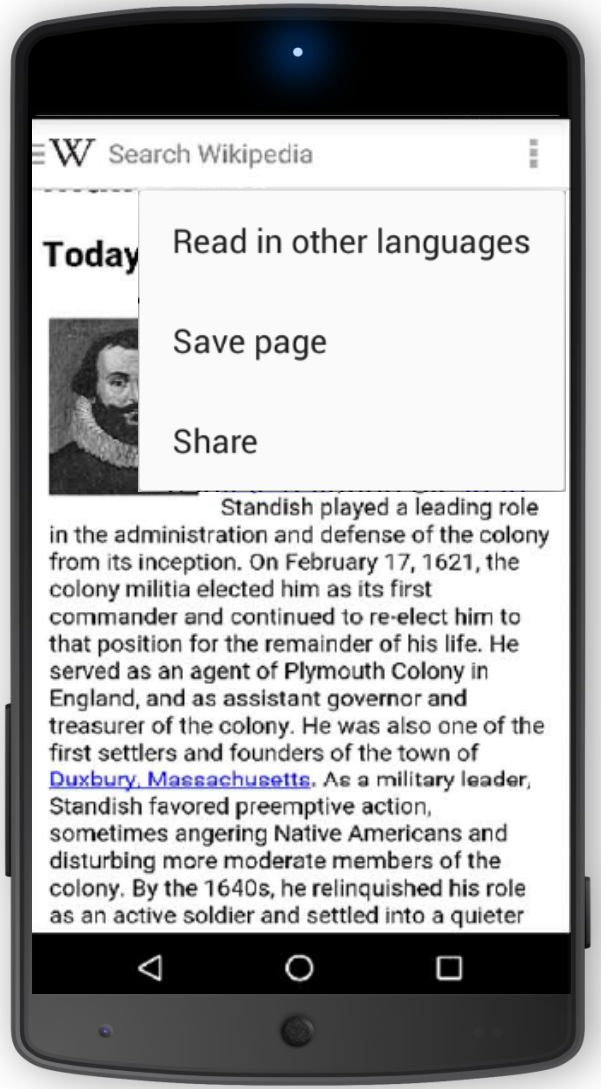


CrowdSeer: Crash Prevention

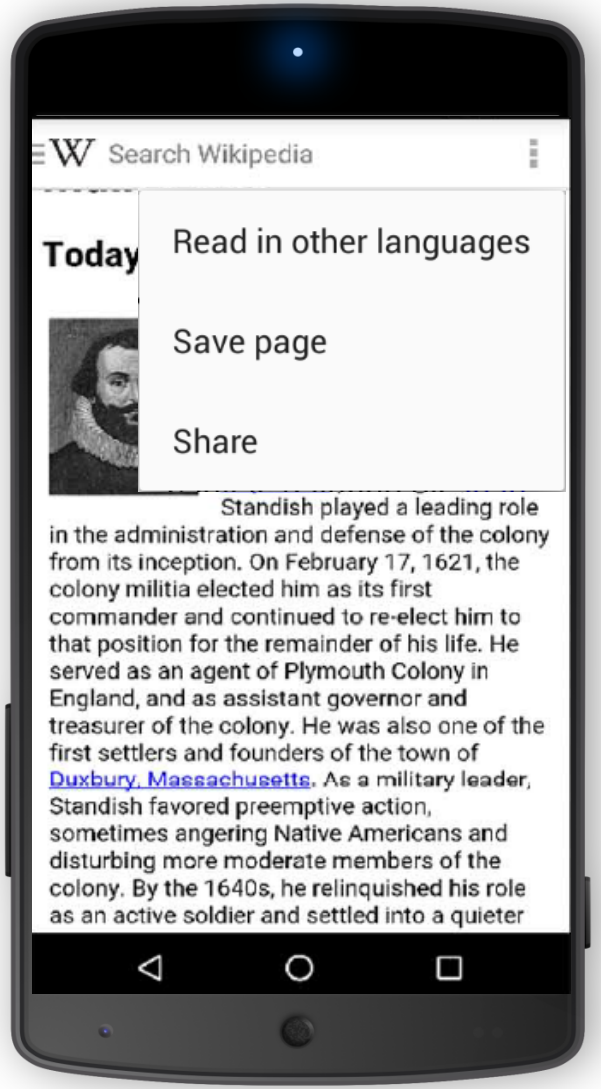
CrowdSeer: Crash Prevention



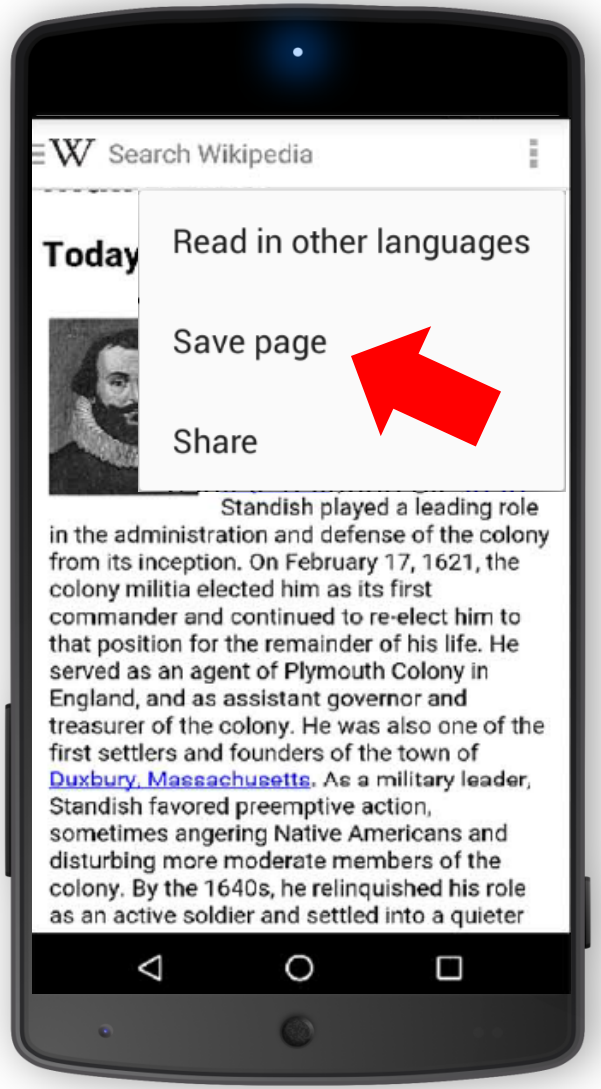
CrowdSeer: Crash Prevention



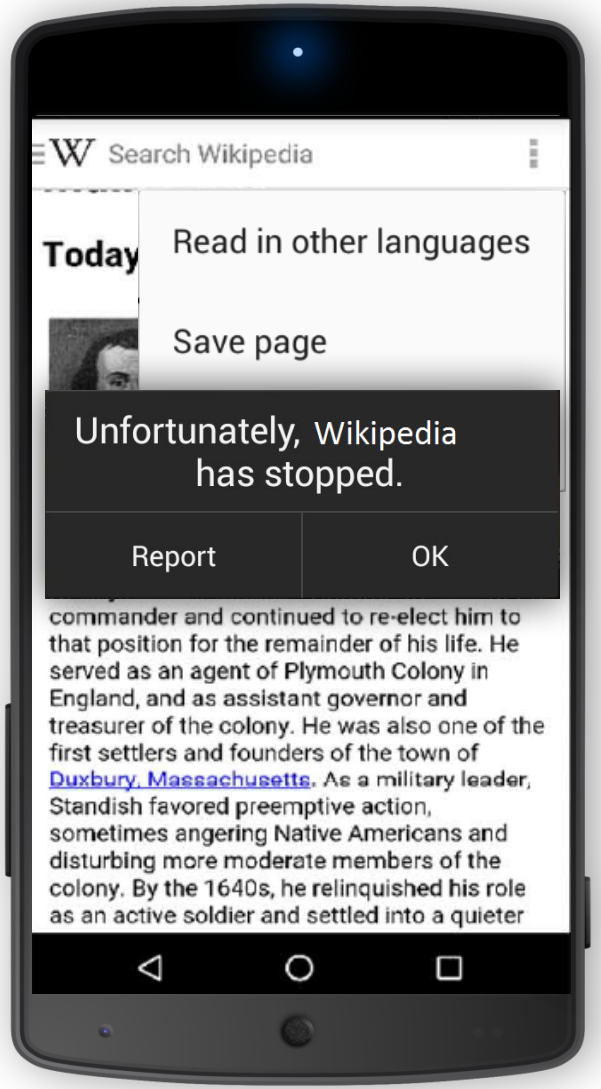
CrowdSeer: Crash Prevention



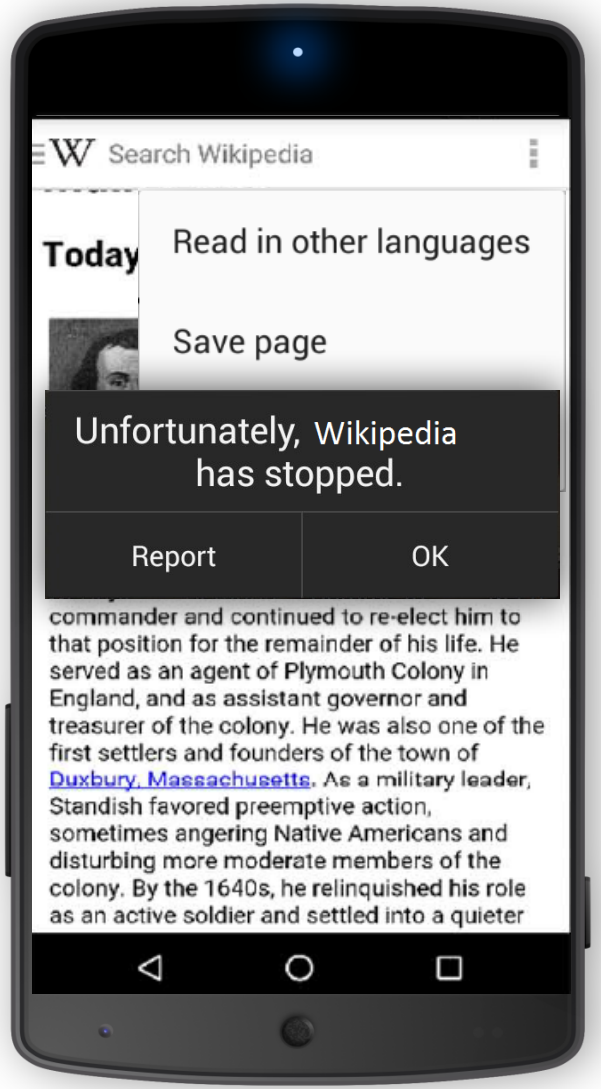
CrowdSeer: Crash Prevention



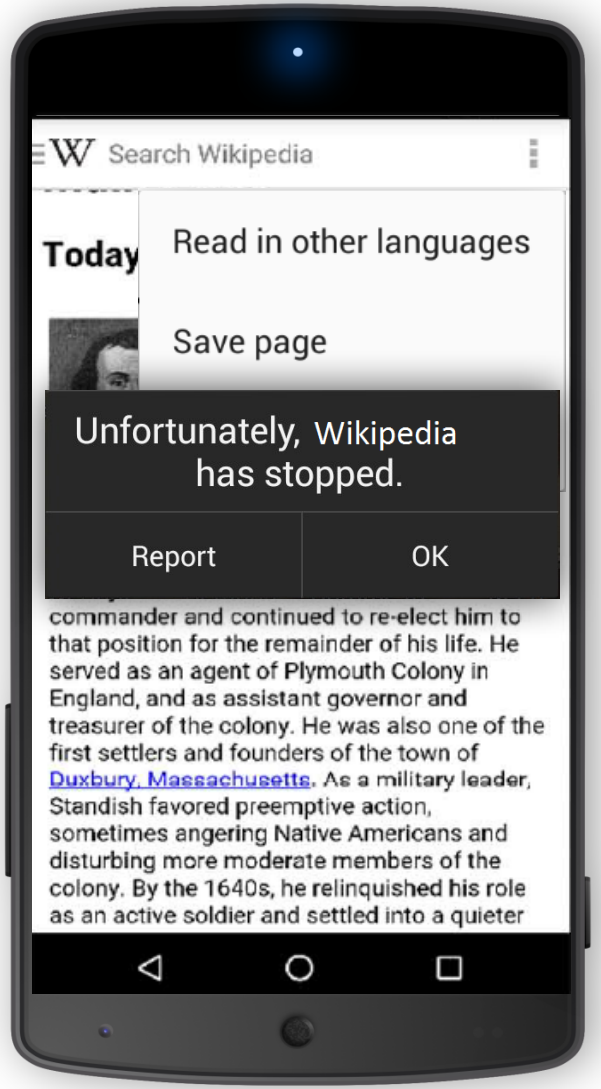
CrowdSeer: Crash Prevention

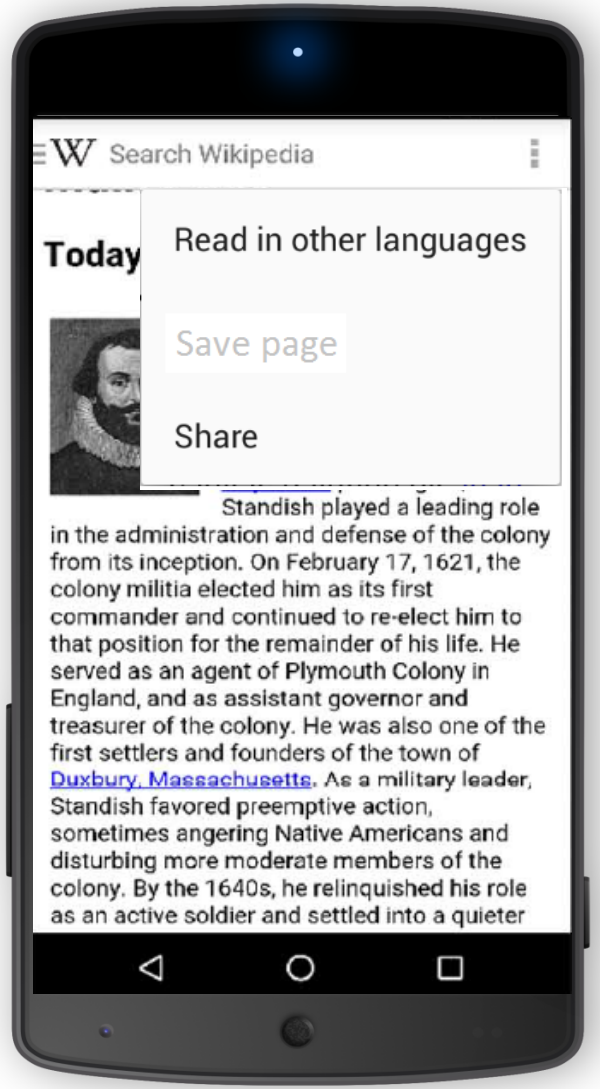


CrowdSeer: Crash Prevention



CrowdSeer: Crash Prevention





Validations

- Lots of interesting insights in the papers based on
  - More than 3,000 apps (some including >100 versions)
  - More than 100 devices
  - More than 500 users (labs, interviews, crowdsourcing)

Questions?

1. M. Gómez, M. Martinez, M. Monperrus, R. Rouvoy: ***When App Stores Listen to the Crowd to Fight Bugs in the Wild***. ICSE-NIER'15
2. M. Gómez, R. Rouvoy, M. Monperrus, L. Seinturier: ***A Recommender System of Buggy App Checkers for App Store Moderators***. MOBILESoft'15
3. G. Hecht, O. Benomar, R. Rouvoy, N. Moha, L. Duchien: ***Tracking the Software Quality of Android Applications Along Their Evolution***. ASE'15
4. M. Gómez, R. Rouvoy, B. Adams, L. Seinturier: ***Reproducing Context-sensitive Crashes of Mobile Apps using Crowdsourced Monitoring***. MOBILESoft'16
5. G. Hecht, R. Rouvoy, N. Moha: ***An Empirical Study of the Performance Impacts of Android Code Smells***. MOBILESoft'16
6. M. Gómez, R. Rouvoy, B. Adams, L. Seinturier: **Mining test repositories for automatic detection of UI performance regressions in Android apps**. MSR'16