

Challenges and Opportunities in Mobile Testing

Alessandra Gorla
IMDEA Software Institute, Madrid, Spain

Intro

B.Sc. and M.Sc. in Milano-Bicocca, Italy

Data-flow testing of Java Applications

Contextual Integration Testing of Classes*

Giovanni Denaro¹, Alessandra Gorla², and Mauro Pezzè^{1,2}

¹ University of Milano-Bicocca, Dipartimento di Informatica, Sistemistica e Comunicazione, Via Bicocca degli Arcimboldi 8, 20126, Milano, Italy
denaro@disco.unimib.it

² University of Lugano, Faculty of Informatics,
via Buffi 13, 6900, Lugano, Switzerland
alessandra.gorla@lu.unisi.ch, mauro.pezze@unisi.ch

Abstract. This paper tackles the problem of structural integration testing of stateful classes. Previous work on structural testing of object-oriented software exploits data flow analysis to derive test requirements for class testing and defines contextual def-use associations to characterize inter-method relations. Non-contextual data flow testing of classes works well for unit testing, but not for integration testing, since it misses definitions and uses when properly encapsulated. Contextual data flow analysis approaches investigated so far either do not focus on state dependent behavior, or have limited applicability due to high complexity. This paper proposes an efficient structural technique based on contextual data flow analysis to test state-dependent behavior of classes that aggregate other classes as part of their state.

1 Introduction

FASE 2008

characterized by classes and objects, which enforce ordering to their internal state. Object-oriented practice, and reduce the impact of some critical those that derive from excessive use of non-local access to hidden details. However, they intro

Search-based Data-flow Test Generation

Mattia Vivanti
University of Lugano
Lugano, Switzerland
mattia.vivanti@usi.ch

Andre Mis · Alessandra Gorla
Saarland University
Saarbrücken, Germany
{amis,gorla}@cs.uni-saarland.de

Gordon Fraser
University of Sheffield
Sheffield, UK
Gordon.Fraser@sheffield.ac.uk

Abstract—Coverage criteria based on data-flow have long been discussed in the literature, yet to date they are still of surprising little practical relevance. This is in part because 1) manually writing a unit test for a data-flow aspect is more challenging than writing a unit test that simply covers a branch or statement, 2) there is a lack of tools to support data-flow testing, and 3) there is a lack of empirical evidence on how well data-flow testing scales in practice. To overcome these problems, we present 1) a search-based technique to automatically generate unit tests for data-flow criteria, 2) an implementation of this technique in the EVOSUITE test generation tool, and 3) a large empirical study applying this tool to the SF100 corpus of 100 open source Java projects. On average, the number of coverage objectives is three times as high as for branch coverage. However, the level of coverage achieved by EVOSUITE is comparable to other criteria, and the increase in size is only 15%, leading to higher mutation scores. These results counter the common assumption that data-flow testing does not scale, and should help to re-establish data-flow testing as a viable alternative in practice.

Keywords—data-flow coverage, search based testing, unit testing

I. INTRODUCTION

Systematic test generation is often driven by coverage criteria based on structural program entities such as statements or branches. In contrast to such structural criteria, data-flow criteria focus on the data-flow interactions within or across

criteria is that if a value in another, then it is these statements to es showed that data-ect-oriented code [4], e usually shorter than -procedural logic, for ded.

ISSRE 2008

statement [8]. This emphasizes the importance of *automated* test generation tools — however, most existing systematic test generation tools target either statement or branch coverage.

A further problem preventing wide-spread adoption of data-flow criteria is a lack of understanding of how well they scale to real world applications. Intuitively, data-flow criteria result in more test objectives to cover, and consequently also more test cases, but the number of infeasible test objectives (i.e., infeasible paths from definitions to uses of the same variable) is also expected to be larger than for simpler structural criteria. However, there simply is not sufficient empirical evidence to decide whether this is a show-stopper in adoption of data-flow testing criteria, or just a minor side effect.

To address these problems, in this paper we present a data-flow test generation technique implemented as an extension of the search-based EVOSUITE [11] tool, which we applied to 100 randomly selected open source Java projects. In detail, the contributions of this paper are as follows:

- We present a search-based technique to generate unit tests for data-flow criteria. This technique uses a genetic algorithm for both, the classical approach of targeting one test objective at a time, as well as the alternative approach of targeting all test objectives at the same time.
- We present an implementation of this technique, extending the EVOSUITE test generation tool to generate test suites targeting all definition-use pairs.
- We present the results of a large empirical study on open source Java applications (the SF100 corpus of classes [12]) in order to shed light on how data-flow testing scales and compares to other criteria in practice.

The results of our experiments indicate that data-flow testing is a viable alternative and does not suffer from scalability

Intro

PhD in Informatics in Lugano, Switzerland

Automatic Workarounds for Web Applications

Automatic Workarounds for Web Applications

Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè*
Faculty of Informatics

Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications

ANTONIO CARZANIGA, University of Lugano, Switzerland
ALESSANDRA GORLA, IMHA Software Institute, Spain
NICOLÒ PERINO, University of Lugano, Switzerland
MAURO PEZZÈ, University of Lugano, Switzerland and University of Milano-Bicocca, Italy

Despite the best intentions, the competences, and the rigorous methods of designers and developers, software is often delivered and deployed with faults. To cope with imperfect software, researchers have proposed the concept of self-healing for software systems. The ambitious goal is to create software systems capable of detecting and responding "autonomously" to functional failures, or perhaps even preventing such failures, to maintain a correct functionality, possibly with acceptable degradation. We believe that self-healing can only be an expression of some form of redundancy, meaning that, to automatically fix a faulty behavior, the correct behavior must be already present somewhere, in some form, within the software system either explicitly or implicitly. One approach is to deliberately design and develop redundant systems, and in fact, the implementation of many fault tolerance techniques (however, and does not always satisfy the time and cost constraints

We observe that modern software systems naturally are redundant, but rather access incrementally as a way to simulate the codes of intrinsic redundancy and we level of self-healing. We first demonstrate that software we develop a way to express and exploit this redundancy, a workaround consists in replacing some failing functionality with a redundant one, but that

particular, we implement with several layers. In this paper, we describe automatic workarounds, and their implementation. We first demonstrate that software we develop a way to express and exploit this redundancy, a workaround consists in replacing some failing functionality with a redundant one, but that

particular, we implement with several layers. In this paper, we describe automatic workarounds, and their implementation. We first demonstrate that software we develop a way to express and exploit this redundancy, a workaround consists in replacing some failing functionality with a redundant one, but that

Automatic Recovery from Runtime Failures

Antonio Carzaniga* Alessandra Gorla† Andrea Mattavelli* Nicolò Perino* Mauro Pezzè*

*University of Lugano
Faculty of Informatics
Lugano, Switzerland

†Saarland University
Computer Science
Saarbrücken, Germany

Abstract—We present a technique to make applications resilient to failures. This technique is intended to maintain a faulty application functional in the field while the developers work on permanent and radical fixes. We target field failures in applications built on reusable components. In particular, the technique exploits the intrinsic redundancy of those components by identifying *workarounds* consisting of alternative uses of the faulty components that avoid the failure. The technique is currently implemented for Java applications but makes little or no assumptions about the nature of the application, and works without interrupting the execution flow of the application and without restarting its components. We demonstrate and evaluate this technique on four mid-size applications and two popular libraries of reusable components affected by real and seeded faults. In these cases the technique is effective, maintaining the application fully functional with between 19% and 48% of the failure-causing faults, depending on the application. The experiments also show that the technique incurs an acceptable runtime overhead in all cases.

1. INTRODUCTION

Software systems are sometimes released and then deployed with faults, and those faults may cause field failures, and this happens despite the best effort and the rigorous methods of developers and testers. Furthermore, even when detected and reported to developers, field failures may take a long time to diagnose and eliminate. As a perhaps extreme but certainly not unique example, consider fault n. 3655 in the Firefox browser,

The problem with these fault-tolerance techniques is that they are expensive and are also considered ineffective due to correlation between faults. Therefore, more recent techniques attempt to avoid or mask failures without incurring the significant costs of producing fully redundant code. Among them, some address specific problems such as inconsistencies in data structures [4], [5], configuration incompatibilities [6], infinite loops [7], security violations [8], and non-deterministic failures [9], [10], while others are more general but require developers to manually write appropriate patches to address application-specific problems [11], [12].

In this paper we describe a technique intended to incur minimal costs and also to be very general. The technique works opportunistically and therefore can not offer strict reliability guarantees. Still, short of safety-critical systems, our goal is to support a wide range of applications to overcome a large class of failures. Similarly to other techniques, the main ingredient we plan to use is redundancy. In particular, we propose to exploit a form of redundancy that is *intrinsic* in modern component-based software systems. We observe that modern software and especially reusable components are designed to accommodate the needs of several applications and therefore to offer many variants of the same functionality. Such variants may be similar enough semantically, but different enough in their implementation, that a fault in one operation can be avoided by executing an alternative variant of the same operation. The automatic selection and execution of a variant (to avoid a failure of a faulty one) is what we refer to as an *automatic workaround*.

Prior work we have developed this notion of automatic workarounds by showing experimentally that such workarounds exist and can be effective in Web applications [13]. We initially focused on Web applications because

Cross-Checking Oracles from Intrinsic Software Redundancy

Antonio Carzaniga
University of Lugano
Switzerland
antonio.carzaniga@usi.ch

Alberto Goffi
University of Lugano
Switzerland
alberto.goffi@usi.ch

Alessandra Gorla
Saarland University
Germany
gorla@st.cs.uni-saarland.de

Andrea Mattavelli
University of Lugano
Switzerland
andrea.mattavelli@usi.ch

Mauro Pezzè
University of Lugano
Switzerland
University of Milano-Bicocca
Italy
mauro.pezze@usi.ch

ABSTRACT

Despite the recent advances in automatic test generation, testers must still write test oracles manually. If formal specifications are available, it might be possible to use decision procedures derived from those specifications. We present a technique that is based on a form of specification but also leverages more information from the system under test. We assume that the system under test is somewhat redundant, in the sense that some operations are designed to behave like others but their executions are different. Our experience in this and previous work indicates that this redundancy exists and is easily documented. We then generate oracles by cross-checking the execution of a test with the same test in which we replace some operations with redundant ones. We develop this notion of cross-checking oracles into a generic technique to automatically insert oracles into unit tests. An experimental evaluation shows that cross-checking oracles, used in combination with automatic test generation techniques, can be very effective in revealing faults, and that they can even improve good hand-written test suites.

Categories and Subject Descriptors

1. INTRODUCTION

Test oracles discriminate successful from failing executions of test cases. Good oracles combine simplicity, generality, and accuracy. Oracles should be simple to write and straightforward to check, otherwise we would transform the problem of testing the software system into the problem of testing the oracles. They should also be generally applicable to the widest possible range of test cases, in particular so that they can be used within automatically generated test suites. And crucially, they should be accurate in revealing all the faulty behaviors (completeness, no false negatives) and only the faulty ones (soundness, no false positives).

Test oracles are often written manually on a case-by-case basis, commonly in the form of assertions, for example JUnit assertions.¹ Such input-specific oracles are usually simple and effective but they lack generality. Writing such oracles for large test suites and maintaining them through the evolution of the system can be expensive. Writing and maintaining such oracles for large *automatically generated* test suites may be practically impossible.

It is possible to also generate oracles automatically, even though research on test automation has focused mostly on supporting the testing process, creating scaffolding, managing regression test suites, and generating and executing test cases, but much less on generating oracles [7, 27]. Most of the work on the automatic generation of oracles is based on some form of specification or model. Such oracles are very generic, since they simply check that the behavior of the system is consistent with the prescribed model. However, their applicability and quality depend on the availability and completeness of the models. For example, specification-based oracles are extremely effective in the presence of precise

FSE 2010

TOSEM 2015

ICSE 2013

ICSE 2014

Intro

Postdoc Saarland University, Germany

Malware detection in Android applications

Checking App Behavior Against App Descriptions

Alessandra Gorla · Ilaria Tavecchia · Florian Gross · Andreas Zeller
Saarland University
Saarbrücken, Germany
{gorla, tavecchia, fgross, zeller}@cs.uni-saarland.de

ABSTRACT

How do we know a program does what it claims to do? After clustering Android apps by their description topics, we identify outliers in each cluster with respect to their API usage. A “weather” app that sends messages thus becomes an anomaly; likewise, a “messaging” app would typically not be expected to access the current location. Applied on a set of 22,500+ Android applications, our CHABADA prototype identified several anomalies; additionally, it flagged 56% of novel malware as such, without requiring any known malware patterns.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Invasive software

General Terms
Security

Keywords
Android, malware detection, description analysis, clustering

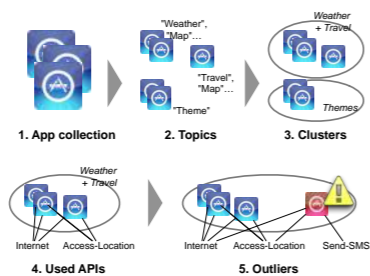


Figure 1: Detecting applications with unadvertised behavior. Starting from a collection of “good” apps (1), we identify their description topics (2) to form clusters of related apps (3). For each cluster, we identify the sensitive APIs used (4), and can then identify outliers that use APIs that are uncommon for that cluster (5).

- An app that sends a text message to a premium number to raise money is suspicious? Maybe, but on Android, this is a legitimate payment method for unlocking game features.
- An app that tracks your current position is malicious? Not if it is a navigation app, a trail tracker, or a map application.

Mining Apps for Abnormal Usage of Sensitive Data

Vitalii Avdiesko¹, Kostasios Karanov¹, Alessandra Gorla², Andreas Zeller², Steven Arzt¹, Siegfried Rasthofer¹, and Eric Bodden^{1†}
¹Saarland University ²IMEIA Software Institute [†]TU Darmstadt ¹Fraunhofer SIT
Saarbrücken, Germany Madrid, Spain Darmstadt, Germany Darmstadt, Germany

Abstract—What is it that makes an app malicious? The important factor is that malicious apps send sensitive data differently from benign apps. To capture such differences, we mined 2,884 benign Android applications for their data flow from sensitive sources, and compare these flows against those found in malicious apps. We find that (a) for every sensitive source, the data ends up in a small number of typical sinks; (b) these sinks differ considerably between benign and malicious apps; (c) these differences can be used to flag malicious apps due to their abnormal data flow; and (d) malicious apps can be identified by their abnormal data flow alone, without requiring known malware samples. In our evaluation, our MUSELOW prototype correctly identified 86.4% of all novel malware, and 95.1% of novel malware leaking sensitive data.

1. INTRODUCTION

Most existing malware detectors work retrospectively, checking an unknown app against features and patterns known to be malicious. Such patterns can either be given explicitly (“Text messages must only be sent after user’s consent”), or induced implicitly from samples of known malware (“This app contains code known to be part of the Trojan”). If a novel app is sufficiently different from known malware, though, this retrospective detection can fail.

In this work, we thus investigate the idea that, given access to a sufficiently large set of “benign” apps, one might be able to detect novel malware not by its similarity with respect to existing malware, but rather through its differences.

TABLE I
FLOWS IN ANDROID TWITTER APP

AccountManager.get()	→ Context.getSystemService()
AccountManager.get()	→ Context.getSystemService()
AccountManager.get()	→ AccountAuthenticator()
AccountManager.get()	→ Log.d()
AccountManager.get()	→ Context.getSystemService()
AccountManager.getPassword()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()
Context.getSystemService()	→ AccountAuthenticator()

TABLE II
FLOWS IN FOX RABBIT (FOXWALL) MALWARE

TelegramManager.getAuthCode()	→ URL.openConnection()
TelegramManager.getAuthCode()	→ URL.openConnection()

In contrast, consider the com.sec.hack0504 malware from the VirusShare database [28]. Table II shows the two flows in that application; they leak the subscriber and device ID to a Web server. Both these flows are very uncommon for benign applications; furthermore, dex2dex does not contain any of the flows that would normally come with apps that use the TelegramManager for legitimate reasons. Thus, dex2dex is an anomaly—not only because it may be similar to known malware, but in particular because its data flows are dissimilar to benignware such as Twitter.

we built a tool called MUSELOW¹ which leverages contextualized [9] static analysis tools to determine such flows for all sensitive Android sources. MUSELOW implements the classifiers, trained on the data flow of benign apps, to statically flag apps with suspicious features. To the best of our knowledge, MUSELOW is the first approach to massively application collections for patterns of “normal” data flow, and these mined patterns to detect malicious behavior.

Search-Based Security Testing of Web Applications

Julian Thomé
Saarland University
Saarbrücken, Germany
sthoem@stud.uni-saarland.de

Alessandra Gorla
Saarland University
Saarbrücken, Germany
gorla@cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

SQL injections are still the most exploited web application vulnerabilities. We present a technique to automatically detect such vulnerabilities through targeted test generation. Our approach uses search-based testing to systematically evolve inputs to maximize their potential to expose vulnerabilities. Starting from an entry URL, our SQUAT prototype systematically creates a web application and generates inputs whose effects on the SQL interaction are assessed at the interface between Web server and database. By evolving these inputs when resulting SQL interactions show low potential, SQUAT exposes vulnerabilities on real-world Web applications within minutes. As a black-box approach, SQUAT requires neither analysis nor instrumentation of server code; however, it does require access to the application’s interface.

Categories and Subject Descriptors
D.1.3 [Testing and Debugging]: Testing tools

General Terms
Security

Keywords



Figure 1: WebChase login page. To bypass authentication, enter 'OR 1=1' as user name, and an arbitrary password.

```
<?php
mysql_select_db('') or die('Database connection failed');
mysql_query('SELECT * FROM players WHERE nick="'.
    $_POST['username']."' AND password='".
    $_POST['password']."' LIMIT 1");
if (mysql_num_rows($result) > 0) {
    //login failed
    exit();
} else {
    //successful login
}
```

Figure 2: WebChase login code, exhibiting authentication bypass via SQL injection.

ICSE 2014

ICSE 2015

SBST 2014

Intro

Assistant professor @ IMDEA software
Madrid, Spain

since January 2015

Automated Test Input Generation for Android: Are We There Yet?

Shauvik Roy Choudhary¹ Alessandra Gorla² Alessandro Orso³
¹Georgia Institute of Technology, USA ²IMDEA Software Institute, Spain ³Georgia Institute of Technology, USA
Email: shauvik@cc.gatech.edu Email: alessandra.gorla@imdea.org Email: orso@cc.gatech.edu

Abstract—Like all software, mobile apps must be adequately tested to gain confidence that they behave correctly. Therefore, in recent years, researchers and practitioners alike have begun to investigate ways to automate app testing. In particular, because of Android's open source nature and its large share of the market, a great deal of research has been performed on input generation techniques for apps that run on the Android operating system. At this point in time, there are in fact a number of such techniques in the literature, which differ in the way they generate inputs, the strategy they use to explore the behavior of the app under test, and the specific heuristics they use. To better understand the strengths and weaknesses of these existing approaches, and get general insight on ways they could be made more effective, in this paper we perform a thorough comparison of the main existing test input generation tools for Android. In our comparison, we evaluate the effectiveness of these tools, and their corresponding techniques, according to four metrics: ease of use, ability to work on multiple platforms, code coverage, and ability to detect

significant differences from Java bytecode, there exist multiple frameworks that can transform Dalvik bytecode into formats that are more familiar and more amenable to analysis and instrumentation (e.g., Java Bytecode [2], Jimple [3], and smali [4]). For all these reasons, there has been a great deal of research in static analysis and testing of Android apps. In the area of testing, in particular, researchers have developed techniques and tools to target one of the most expensive software testing activities: test input generation. There are in fact a number of these techniques in the literature nowadays, which differ in the way they generate inputs, the strategy they use to explore the behavior of the app under test, and the specific heuristics they use. It is however still unclear what are the strengths and weaknesses of these different approaches, how effective they are in general and with respect to one another, and if and how they could be improved.

To answer these questions, in this paper we present a comparative study of the main existing test input generation techniques for Android. The goal of the study is twofold. The first goal is to assess these techniques (and corresponding tools) and understand how they compare to one another and which may be more suitable in which context (e.g., type of app). The second goal is to get a better understanding of the general skills involved in test input generation for Android and identify ways in which existing techniques can be improved. In our comparison, we use the following metrics: ease of use, ability to work on multiple platforms, code coverage, fault detection, and the amount of manual effort involved in their use. Although this is a very practical comparison, and one that normally receives only limited attention

under submission

An Evolutionary Approach for Dynamic Invariant Generation at the Unit-Level

Juan P. Galeotti¹ Alessandra Gorla² Diego Garbervetsky¹ Gordon Fraser³ Andreas Zeller⁴
¹Saarland University, Saarbrücken, Germany ²IMDEA Software Institute, Madrid, Spain ³Departamento de Computación, FCEyN, Buenos Aires, Argentina ⁴University of Sheffield, Sheffield, UK

ABSTRACT

Dynamic invariant detection allows mining of specifications from existing systems, but the quality of the resulting invariants depends on the executions observed. Unobserved behavior cannot be captured by invariants and leads to spurious invariants. Although this can be countered by producing additional executions with automated test generation techniques, the number of possible executions is typically infinite, and invariant detection techniques do not scale well. Consequently, a good set of tests for invariant mining needs to be small, but still result in invariants that are representative enough to cover as many as possible assertions. Model checking addresses exactly this: it counts the number of executions covered by an invariant. In this paper, we use model checking to guide test generation towards finding a representative test set that covers the code and results in

```
DefaultStructure.StackOf.get(163) : DDTM
  invoke int of ( 0, 124, 251 )

DefaultStructure.StackOf.get(163) : DDTM
  invoke int of 340
```

Figure 1. Two example DAIKON invariants derived for the same method, but with different test suites.

However, a general limitation of dynamic invariants is that they apply only to the executions observed so far, and thus heavily depend on the quantity and the quality of the given executions.

Invariants can be generated automatically using test generation techniques such as random testing (e.g., RANCOOP [7]), dynamic symbolic execution (e.g., PEX [7]), or search-based testing (e.g., EvoSuite [7]). If one were to generate all the executions of a program, then an invariant detector would only infer sound invariants (i.e., there would exist no execution that would violate the invariant). However, generating all executions is neither feasible, nor would invariant detection scale up to the task.

to be submitted

Interested in internships
short visits
giving talks??



Contact me

<https://www.software.imdea.org/~alessandra.gorla>
alessandra.gorla@imdea.org

About this talk

- *intro to Android*
- *state of the art in Android testing*
- *open challenges and opportunities ahead*

The mobile market and the Android ecosystem

371K
Babies born per day



378K
iPhones sold per day

562K
iOS devices



700K
Android devices
activated per day



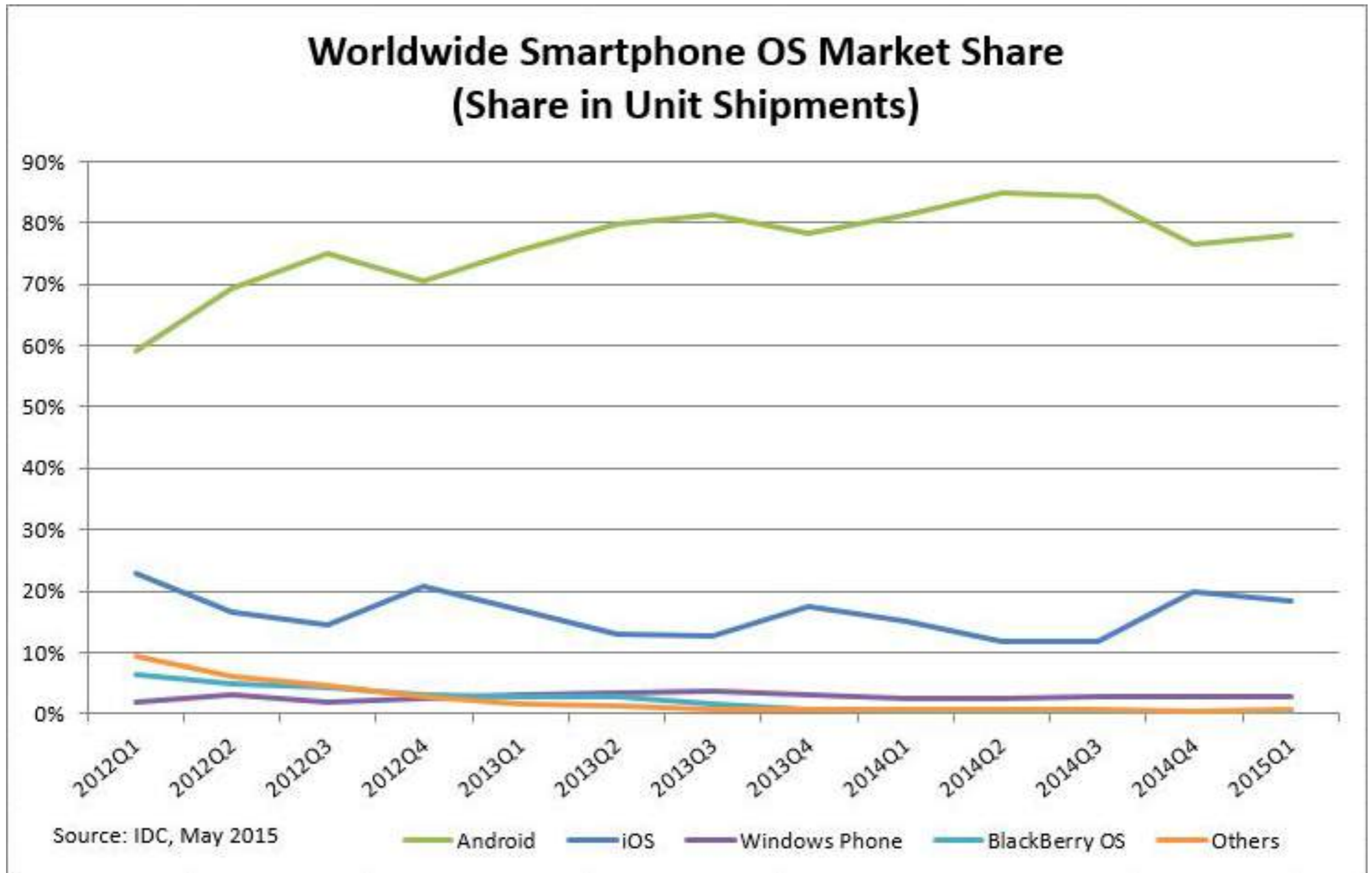
200K
Nokia smartphones



143k
Blackberry devices

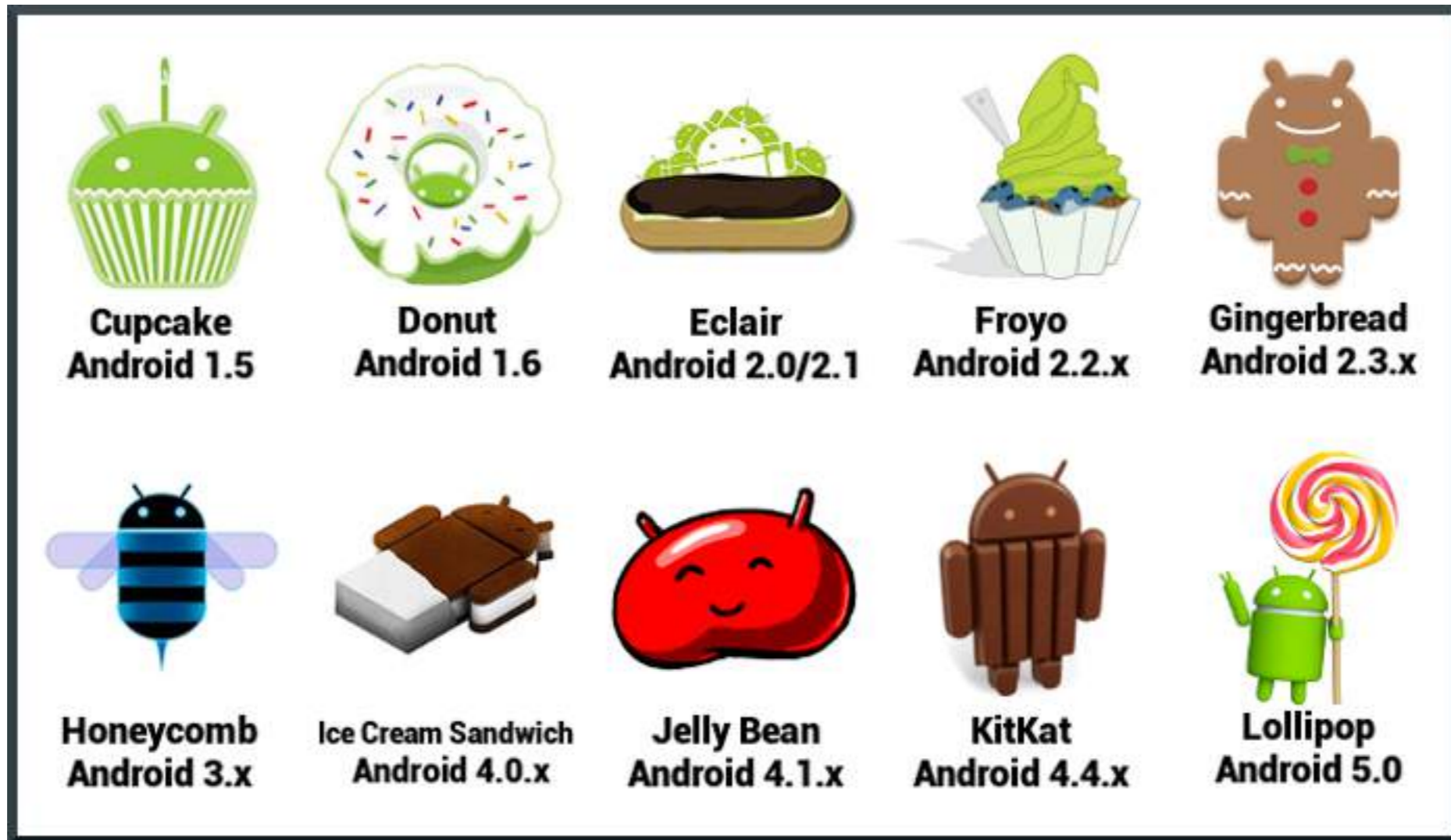
The growth of the mobile market is impressive

Mobile market



Android history

2009

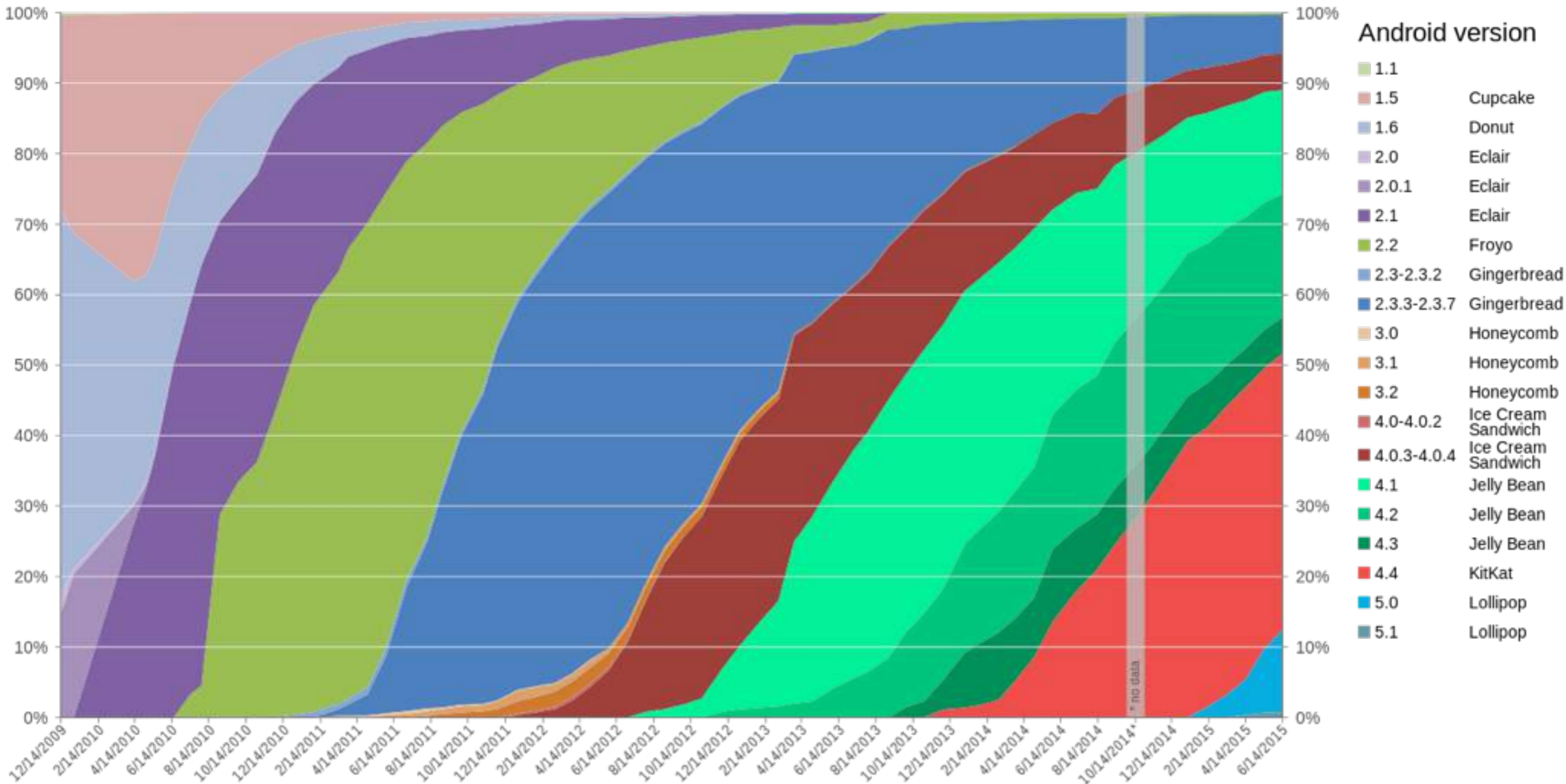


M

Android devices



Release adoption



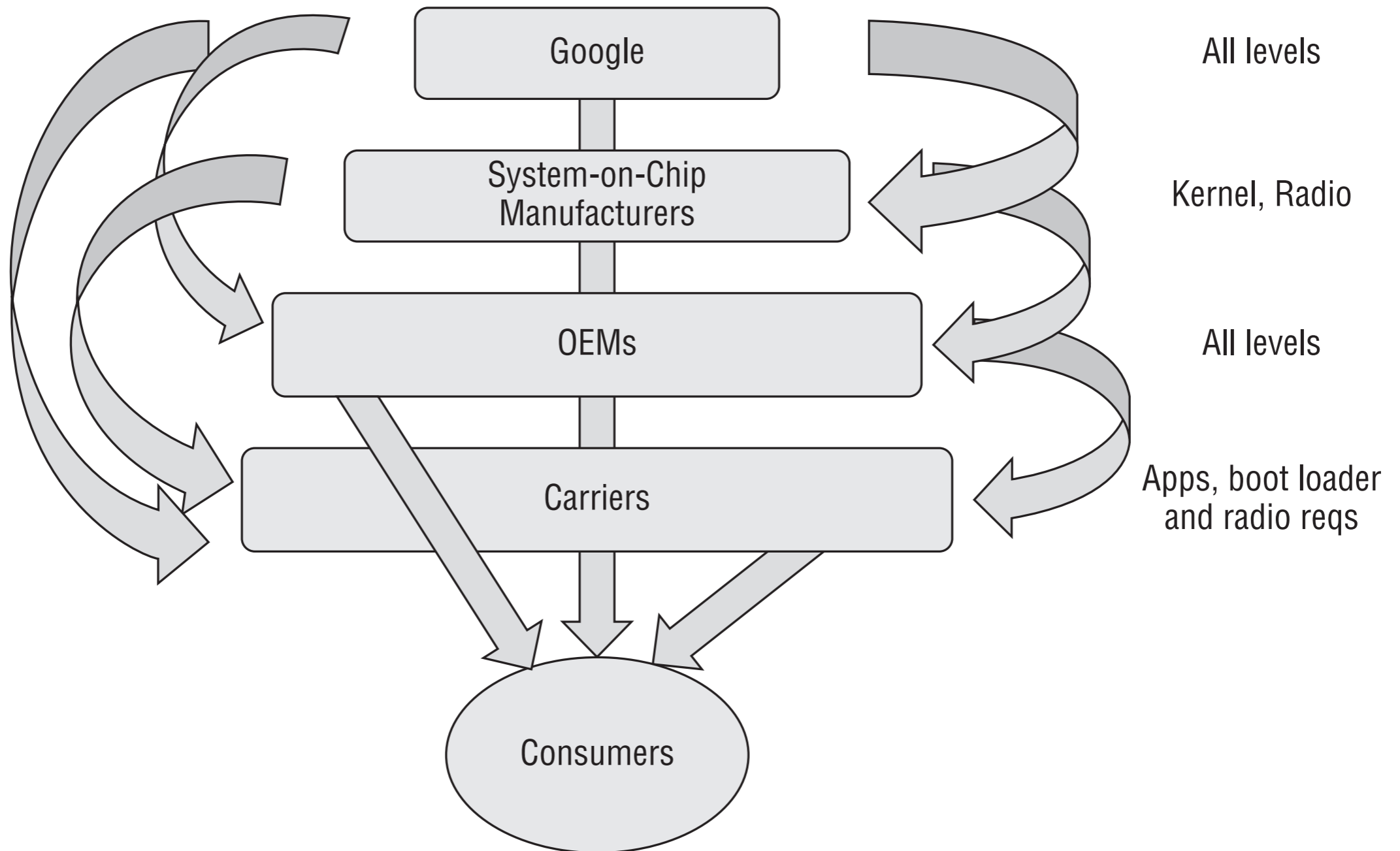
Open source culture

- Android operating system is build upon many different open source components.
 - libraries
 - Linux kernel
 - user interface
 - applications

... but

- There are also closed source components
 - boot loaders
 - peripheral firmware
 - radio components
 - Applications
- And changes in Android are not made available to the public immediately

Android stakeholders



Developers

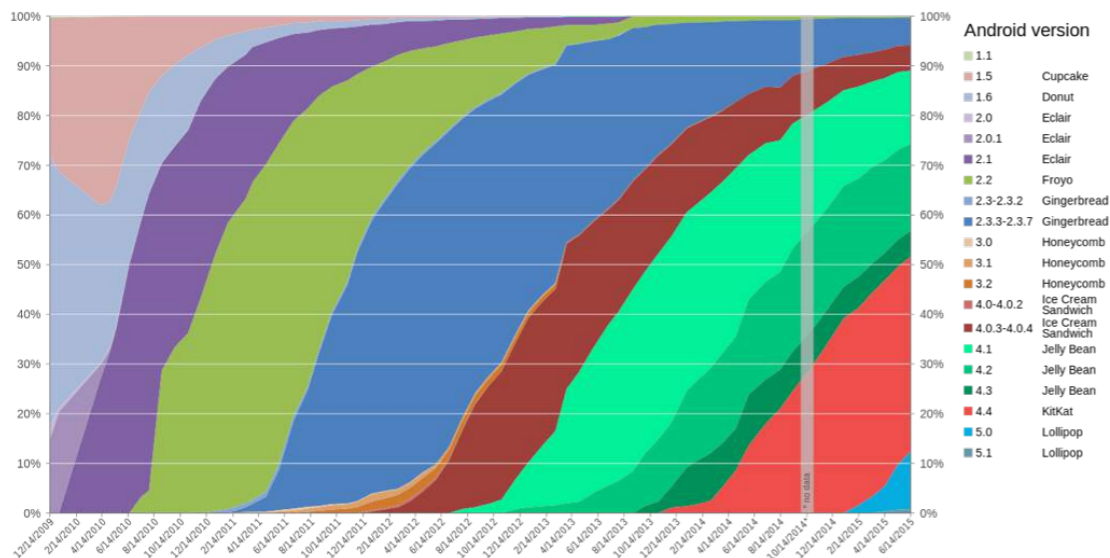
- Developers may contribute to the Android platform.
 - Code review process by Google before including external code.
- Most of external developers contribute by writing apps (through SDK and APIs)
 - Automated analysis before publishing an app in the store.
 - Ranking and report system for further quality

Ecosystem complexity

- fragmentation in hardware
- fragmentation in software
- customization

Issues for quality assurance?

QA issues



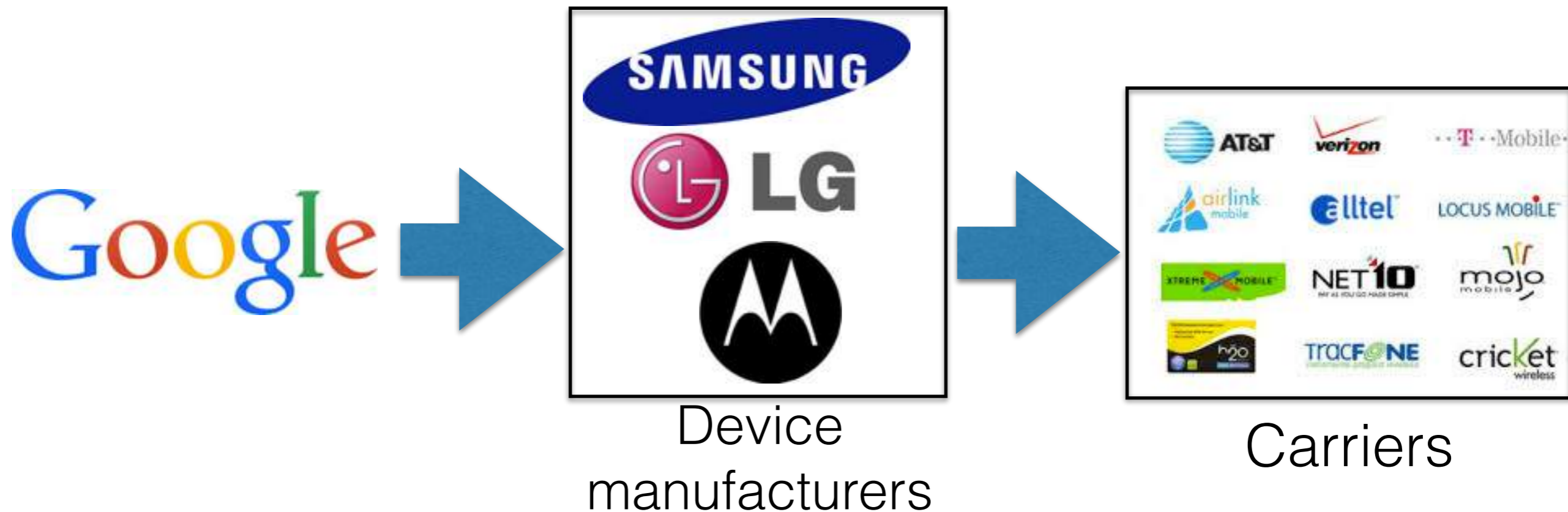
~4 OS releases

X



+1000 devices

Security issues



- Updates might take a long time before being propagated to carrier specific devices.
- Security issues may be fixed after a long time (or even never).

Pros

Cons

Security issues are often specific to hw and sw configurations.

Fragmentation makes it hard to develop security attacks that are valid for most devices.

Security issues detected in the main Android components might take a long time before they are fixed on all devices

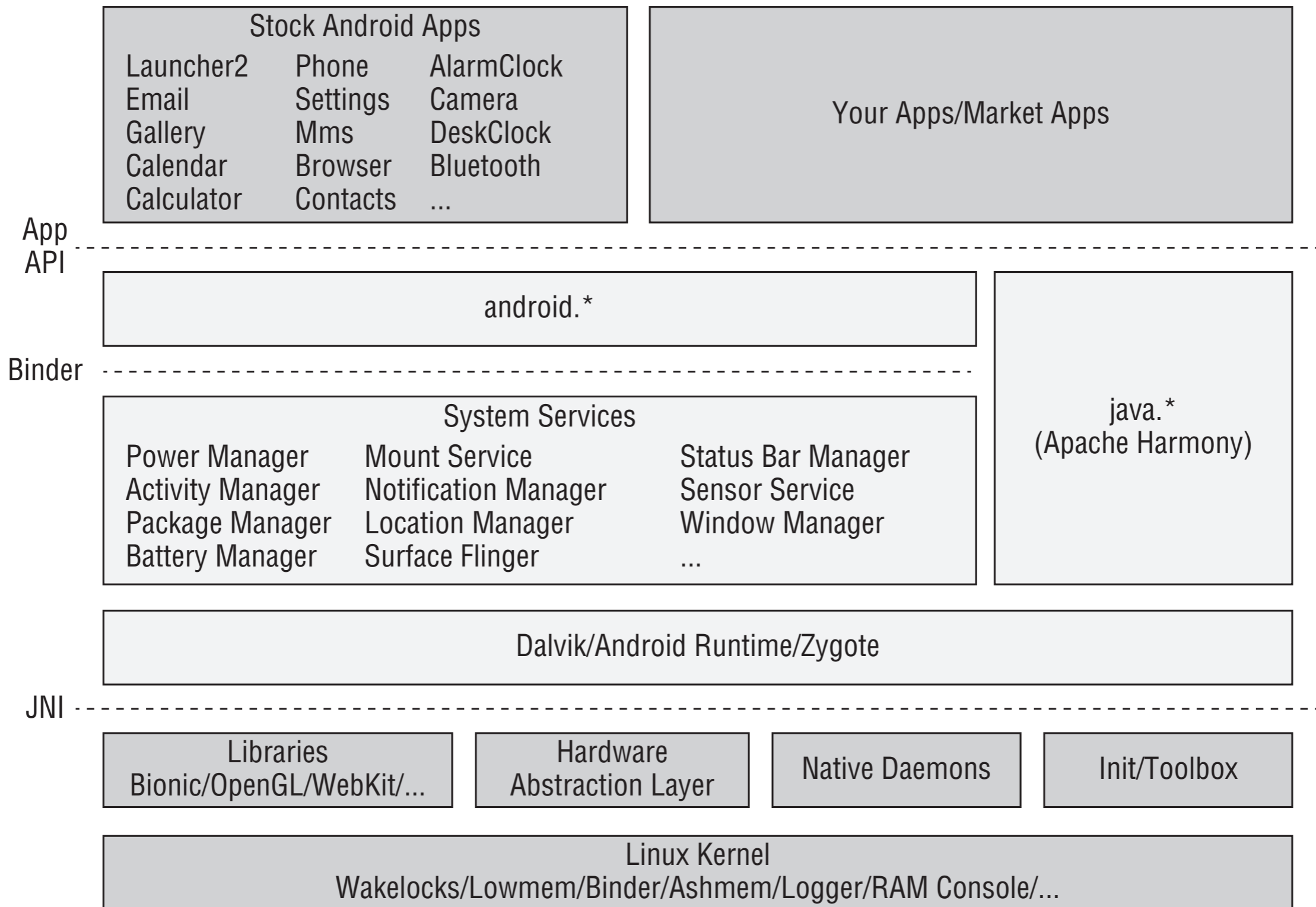


Update mechanisms

- Updates to Android are pushed to Nexus phones directly by Google. Days-weeks between security issue report and pushing a fix.
- For other devices it takes longer. Months-years or even never.
- Almost no back-porting (i.e. applying a fix to older versions of the system).
- Updates to apps are easier. Done directly by app developers through the Google store.

Android architecture

Android components



Dalvik VM

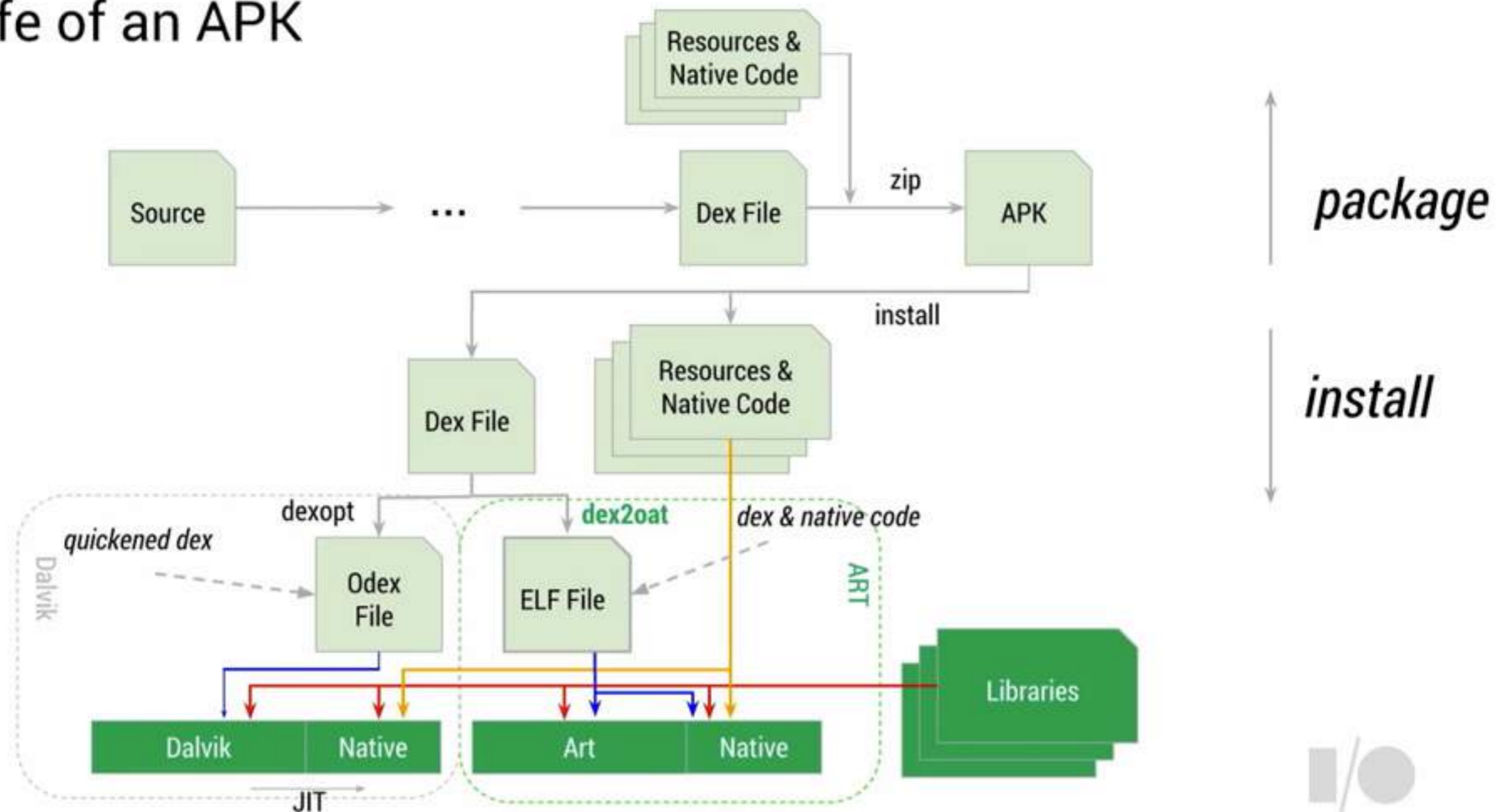
- Specifically designed to provide an efficient abstraction layer to the underlying OS
 - register-based VM
 - interprets Dalvik Executable (DEX bytecode format)
 - relies on functionalities provided by a number of supporting native code libraries

Android RunTime

- although..
- Google recently introduce a new runtime environment: ART (Android RunTime)
 - experimental in Android 4.4 (KitKat)
 - default in Android Lollipop
 - main advantage: performance. Instead of Just In Time compiler, it now compiles Ahead Of Time

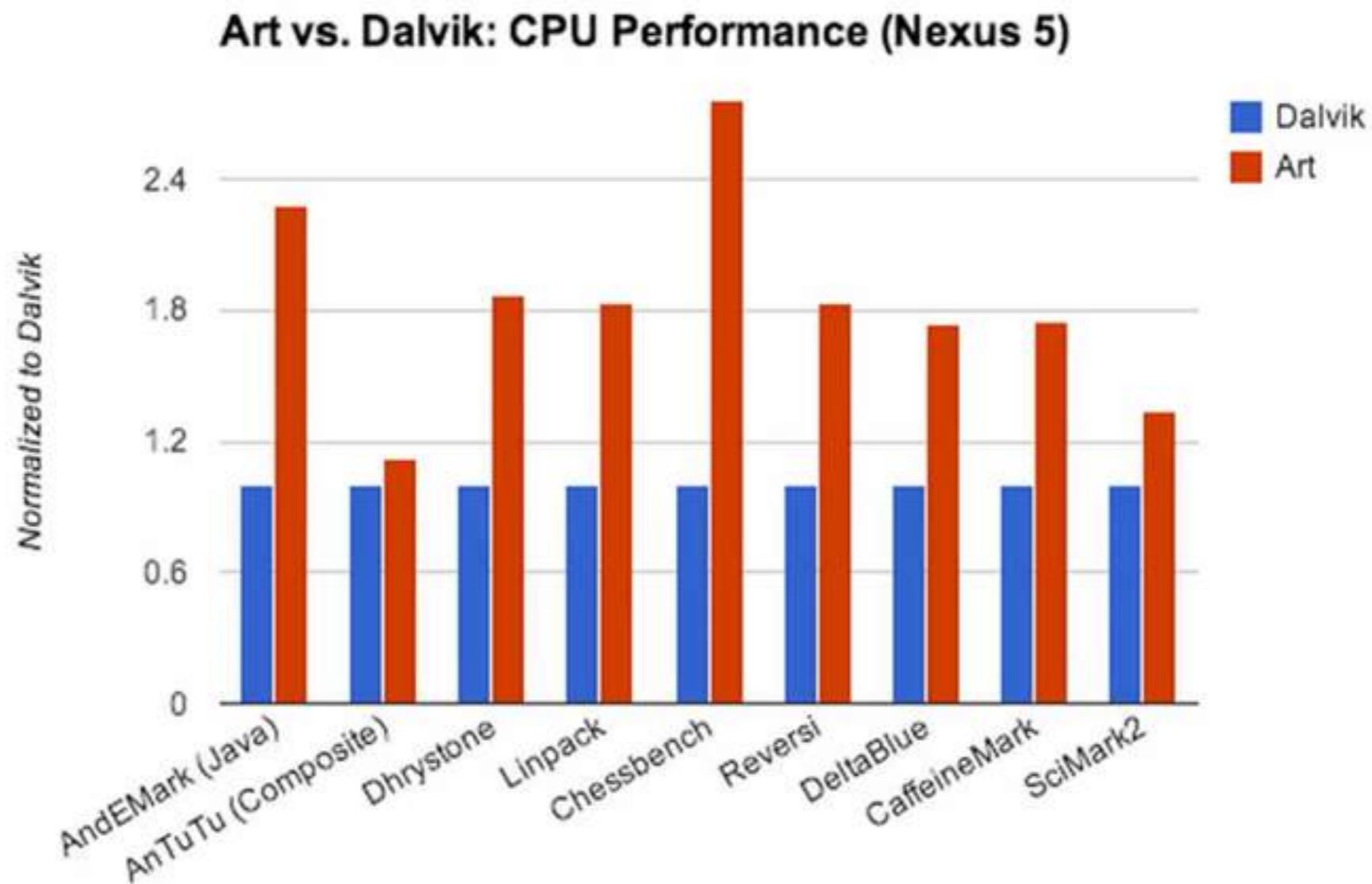
Android Runtime

The life of an APK



Android RunTime

Performance Boosting Thing, realized



Zygote

- Daemon responsible of launching apps.
- Forks a new process for each app.

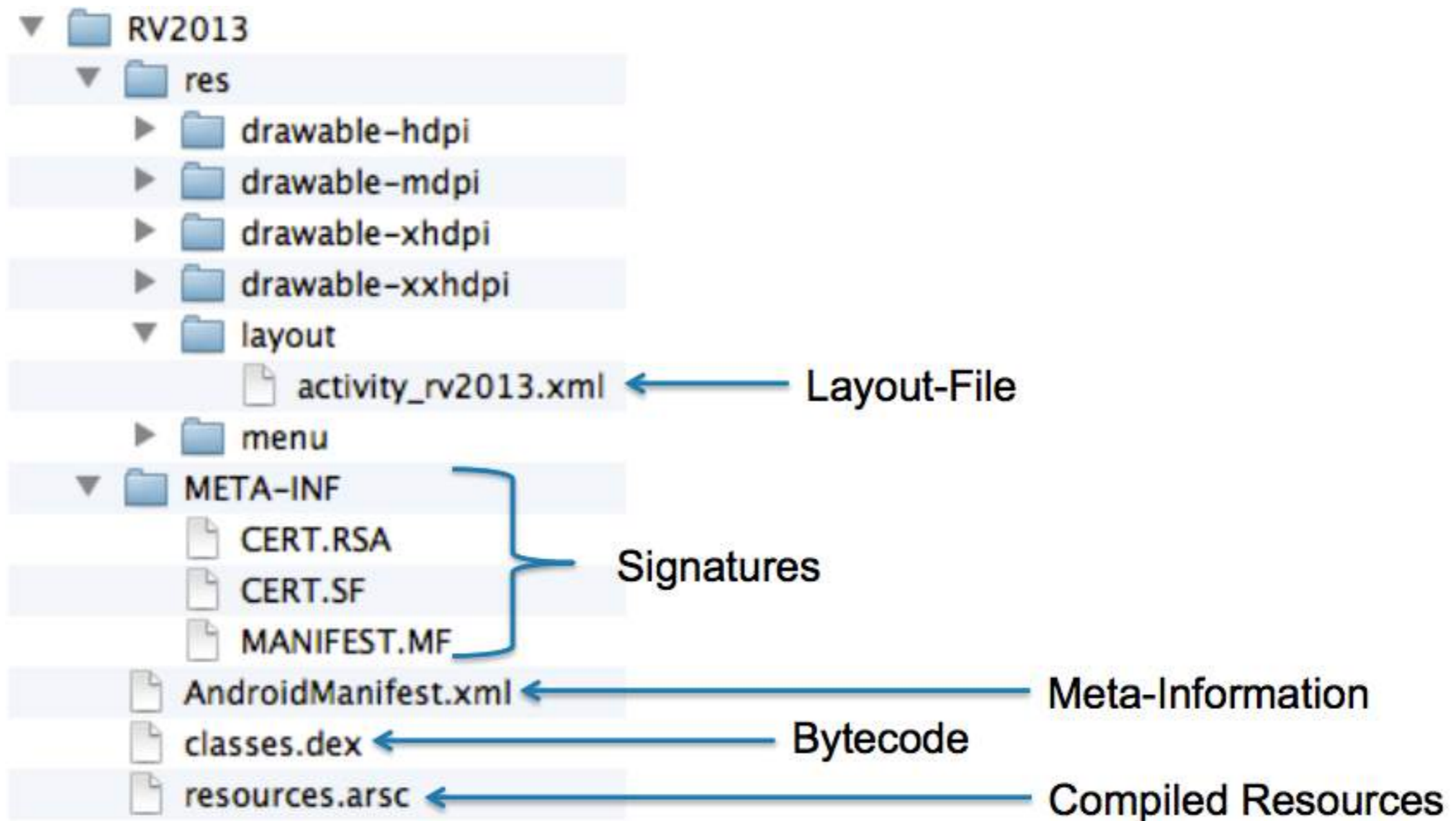
User-space native code components

- Include system services and libraries
 - they communicate with the kernel-level services and drivers.
 - facilitate the low-level operations

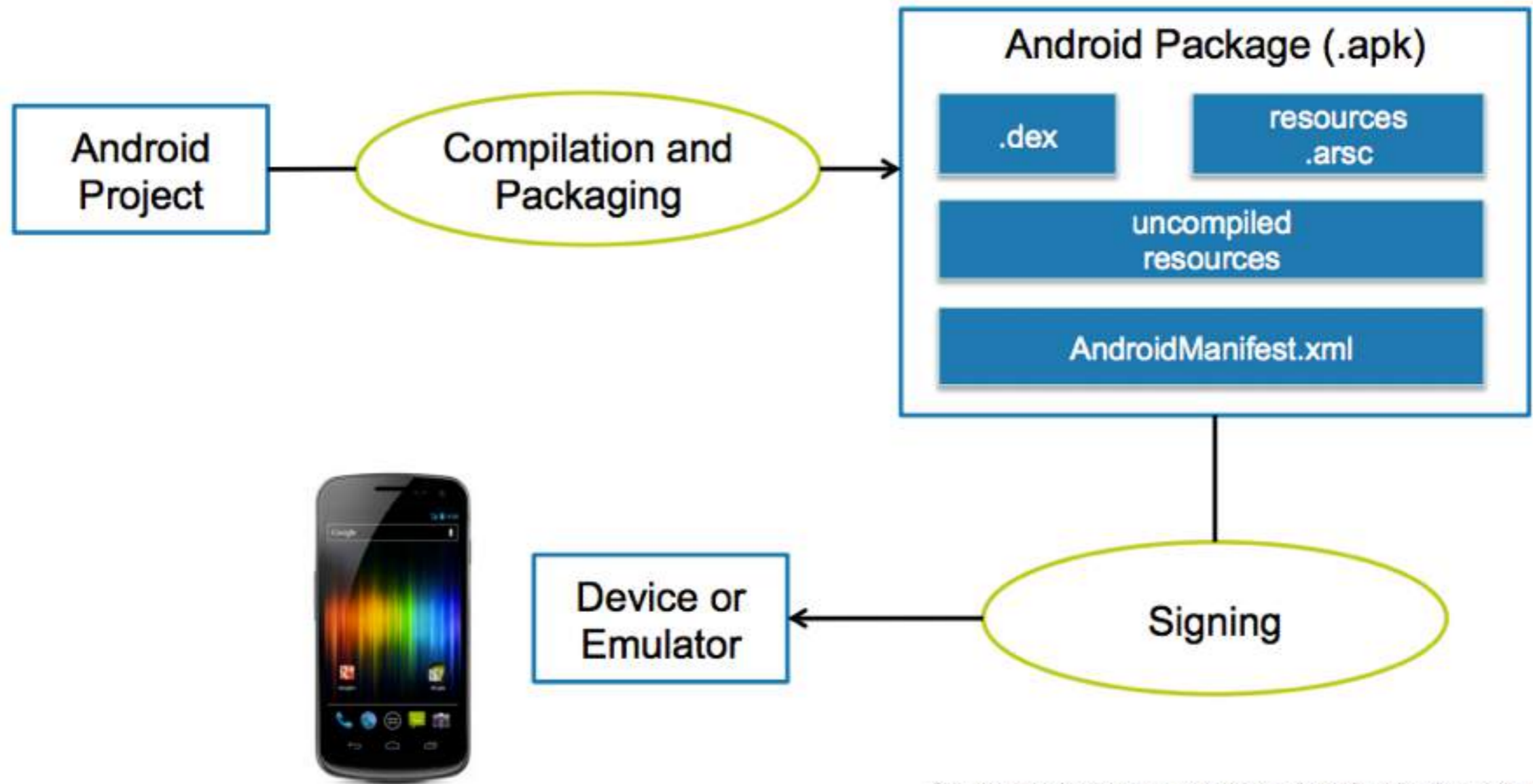
Linux Kernel

- Android made numerous additions and changes to the kernel.
- provide additional functionalities such as
 - camera access
 - wi-fi
 - binder driver (for inter-processes communication)

Main components of an Android app



APK building process



Android Manifest

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.novaapps.findevents" android:versionCode="1"
4     android:versionName="1.0"
5     android:installLocation="preferExternal">
6     <uses-sdk android:minSdkVersion="4" />
7     <supports-screens
8         android:largeScreens="true"
9         android:normalScreens="true"
10        android:smallScreens="true"
11        android:resizeable="true"
12        android:anyDensity="true" />
13
14    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
15    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
16    <uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
17    <uses-permission android:name="android.permission.READ_PHONE_STATE" />
18    <uses-permission android:name="android.permission.INTERNET" />
19    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
20
21    <application android:icon="@drawable/icon" android:label="@string/app_name">
22        <activity android:name=".FindEventsGADroidActivity"
23            android:label="@string/app_name" android:configChanges="orientation|keyboardHidden">
24            <intent-filter>
25                <action android:name="android.intent.action.MAIN" />
26                <category android:name="android.intent.category.LAUNCHER" />
27            </intent-filter>
28        </activity>
29
30        <activity android:name="com.phonegap.DroidGap" android:label="@string/app_name"
31            android:configChanges="orientation|keyboardHidden">
32            <intent-filter>
33                <action android:name="android.intent.action.MAIN" />
34                <category android:name="android.intent.category.LAUNCHER" />
35            </intent-filter>
36        </activity>
37    </application>
38 </manifest>
```

Unique package name

List of activities, services...

Permission definitions

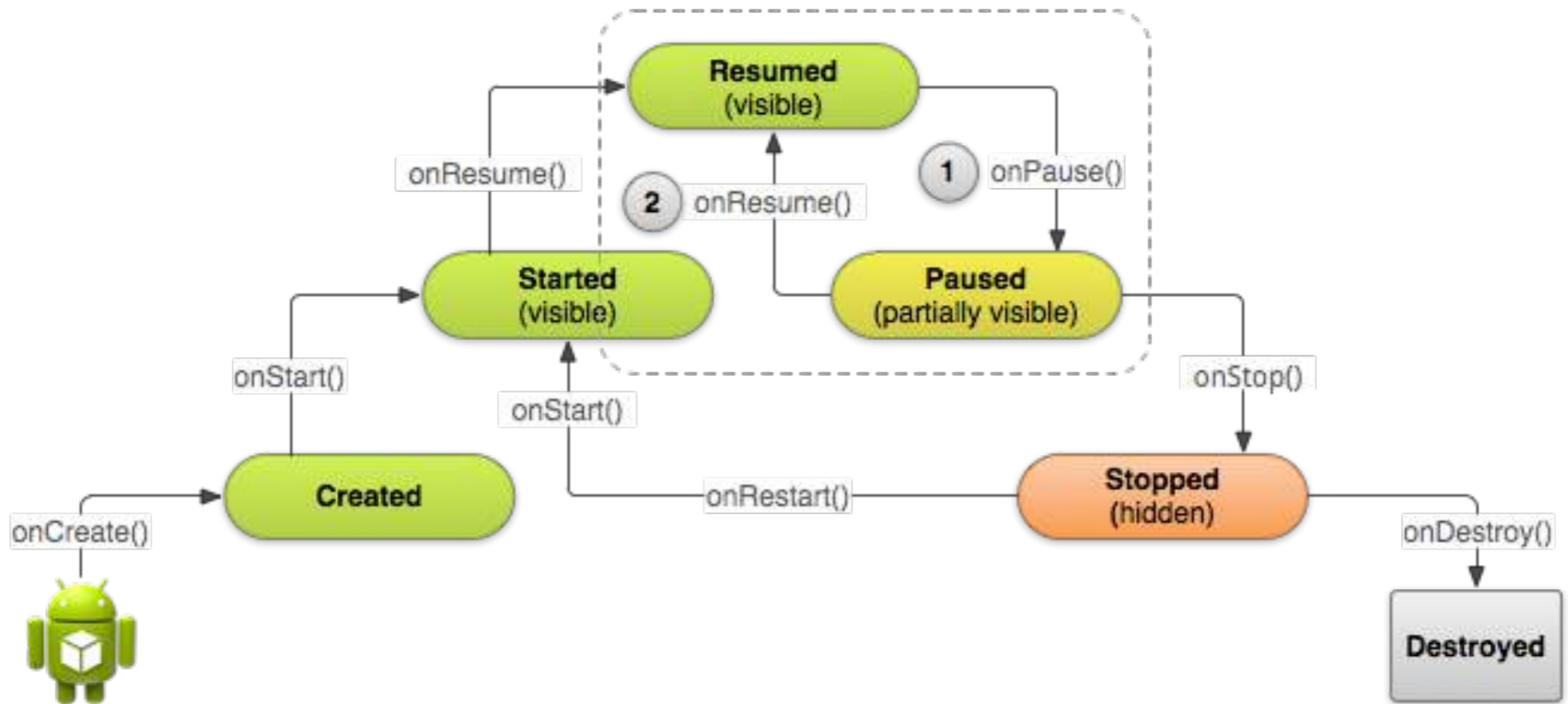
External libraries

shared UID information
preferred installation location

Activities

- In essence it is the UI.
- An activity consists of a window along with several other UI elements.
- Activities are managed by the activity manager service (which also processes intents that are sent to invoke activities).

Activity life cycle



Services

- Application components without UI that run in the background.
- For example, SmsReceiver or BluetoothService
- Services can typically be stopped, started or bound all by way of Intents.

Intents

- Intents are the key part of inter-app communications.
- they are message objects that contain information about an operation to be performed (e.g. make a phone call)
- Intent can also be *implicit*, when they do not have a specific destination.

Broadcast Receivers

- Another component of the IPC.
- Commonly found where applications want to receive an implicit intent matching certain criteria (e.g. receive a SMS).
- They can also be registered at runtime (i.e. not necessarily in the Android Manifest)

Content providers

- Act as a structured interface to common shared data stores (typically SQLite).
- E.g. Contacts and Calendar providers manage centralized repositories with different entries
- Applications may have they own content provider, and may expose it to other apps.

Android security model

Security Boundaries

- Places in the system where the level of trust differs on either side
 - Boundary between kernel-space and user-space.
 - Code in kernel space is trusted to perform low-level operation and access physical memory.
 - Code in user-space cannot access all the memory.

Permissions in Android

- Android OS uses two separate but cooperative permission models
 - Low level: Linux kernel enforces permissions using users and groups (inherited by Linux)
 - Low level permission system is usually referred to as the Android sandbox.
 - High level: app permissions, which limit the abilities of Android apps.
 - The Android runtime/Dalvik VM enforce the high level model

Android's sandbox

- Unix-like process isolation
- Principle of least privilege

Android sandbox

- Processes run as separate users and cannot interfere with each other (e.g. send signals or access one another's memory space)
- Unique user IDs for most processes
- Tightly restricted file system permissions

UID's

- Android shares Linux's UID/GID paradigm, but does not have the traditional *passwd* and *group* files for credentials.
- Android defines a map of names to unique identifiers known as Android IDs (AIDs)
- In addition to AIDs, Android uses supplementary groups to enable processes to access shared/protected resources (e.g. `sdcard_rw`)

At runtime

- When apps execute their UID, GID and supplementary groups are assigned to a newly created process.
- Running under unique UID and GID enables the operating system to enforce lower-level restrictions in the kernel
- Inter-app interaction is possible, and it is controlled by the runtime environment.

output of PS command

app_16	4089	1451	304080	31724	...	S	com.htc.bgp
app_35	4119	1451	309712	30164	...	S	com.google.android.calendar
app_155	4145	1451	318276	39096	...	S	com.google.android.apps.plus
app_24	4159	1451	307736	32920	...	S	android.process.media
app_151	4247	1451	303172	28032	...	S	com.htc.lockscreen
app_49	4260	1451	303696	28132	...	S	com.htc.weather.bg
app_13	4277	1451	453248	68260	...	S	com.android.browser

File system permissions

```
root@android:/ # ls -l /data/data
drwxr-x--x u0_a3    u0_a3 ... com.android.browser
drwxr-x--x u0_a4    u0_a4 ... com.android.calculator2
drwxr-x--x u0_a5    u0_a5 ... com.android.calendar
drwxr-x--x u0_a24   u0_a24 ... com.android.camera
...
drwxr-x--x u0_a55   u0_a55 ... com.twitter.android
drwxr-x--x u0_a56   u0_a56 ... com.ubercab
drwxr-x--x u0_a53   u0_a53 ... com.yougetitback.androidapplication.virgin.
mobile
drwxr-x--x u0_a31   u0_a31 ... jp.co.omronsoft.openwnn
```

Android permissions

- Permissions are required for:
 - System API calls
 - Database operations (content providers)
 - Inter Process Communications (send and receive *Intents*)

Application's permissions

- Extracted from the application's manifest at install time by the PackageManager and stored in /data/system/packages.xml

```
<package name="com.android.chrome"
codePath="/data/app/com.android.chrome-1.apk"
nativeLibraryPath="/data/data/com.android.chrome/lib"
flags="0" ft="1422a161aa8" it="1422a163b1a"
ut="1422a163b1a" version="1599092" userId="10082"
installer="com.android.vending">
<sigs count="1">
<cert index="0" />
</sigs>
<perms>
<item name="com.android.launcher.permission.INSTALL_SHORTCUT" />
<item name="android.permission.NFC" />
...
<item name="android.permission.WRITE_EXTERNAL_STORAGE" />
<item name="android.permission.ACCESS_COARSE_LOCATION" />
...
<item name="android.permission.CAMERA" />
<item name="android.permission.INTERNET" />
...
</perms>
</package>
```

API permissions

- e.g. READ_PHONE_STATE: Read only access to the phone state.
- An app that requires this permission would therefore be able to call a variety of methods related to querying the phone state

`getDeviceSoftwareVersion()`

`getDeviceId()`

IPC permissions

- e.g. `CALL_PHONE`: permission to start a phone call
- An application requires permissions to communicate with another app.

```
Intent intent = new Intent(Intent.ACTION_CALL, Uri.parse(...));  
startActivity(intent);
```

Content Provider permissions

- e.g. READ_CONTACTS, WRITE_CONTACTS: read or write access to the contacts provider.
- An application requires permissions to access a resource at a given URI

State of the art in test
input generation for
Android

Inputs?

Android apps are highly interactive and event driven.

UI events (clicks, longclicks, text)

System events (sms received...)

Environment

Different strategies

Random

Systematic

Model-based (static - dynamic)

Search-based algorithms

Symbolic-execution

Many useful available frameworks!

Useful Frameworks

- *UI automation*
 - Robotium
 - Espresso
 - UI automator
- *Static analysis*
 - DARE
 - Dex disassemblers
 - Soot and Flowdroid

Robotium



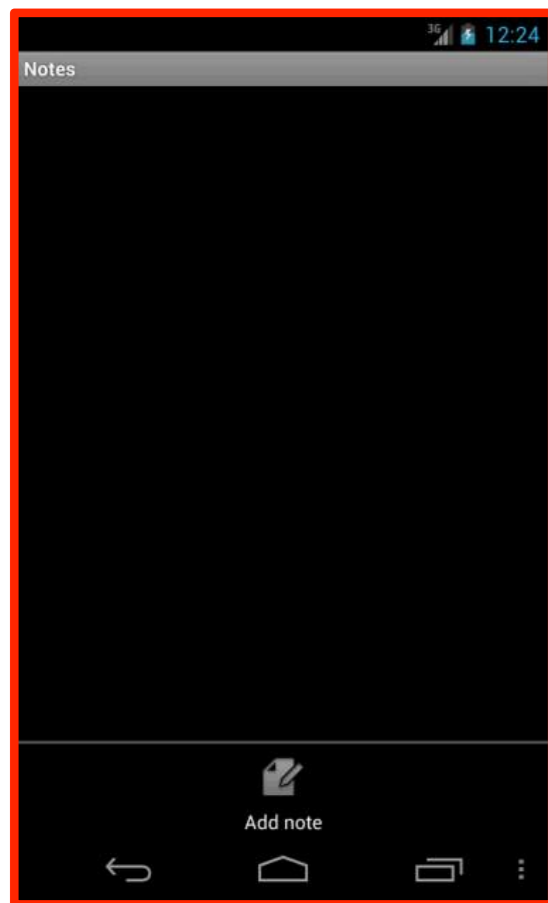
An open source test framework

Used to write black or white box tests

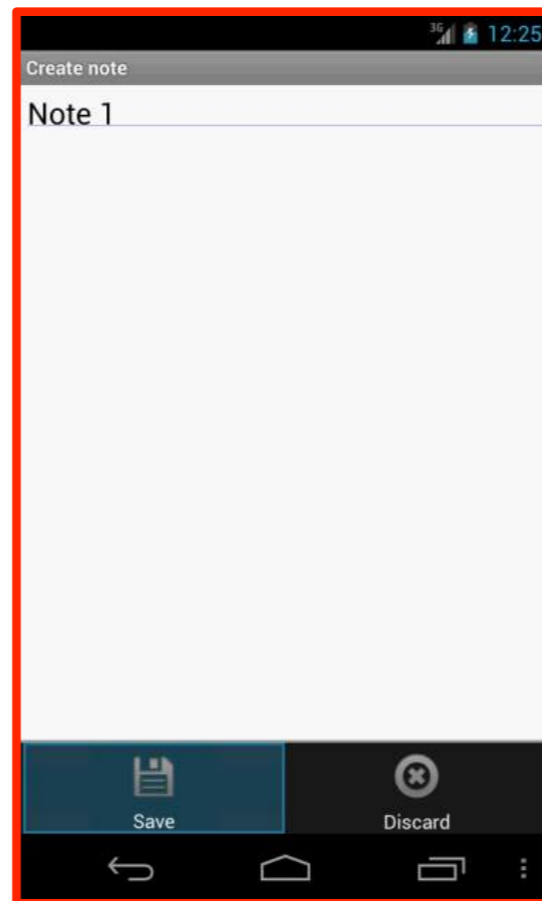
Tests can be executed on an Android Virtual Device (AVD) or a real device

Built on Java and Android JUnit Test Framework

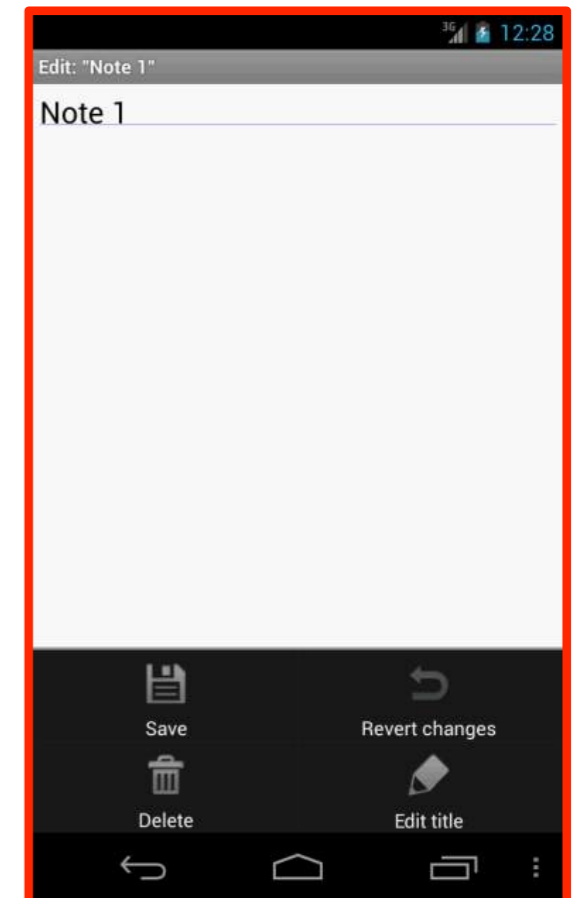
Notepad with Robotium



Add note



Save note



Edit note

Robotium

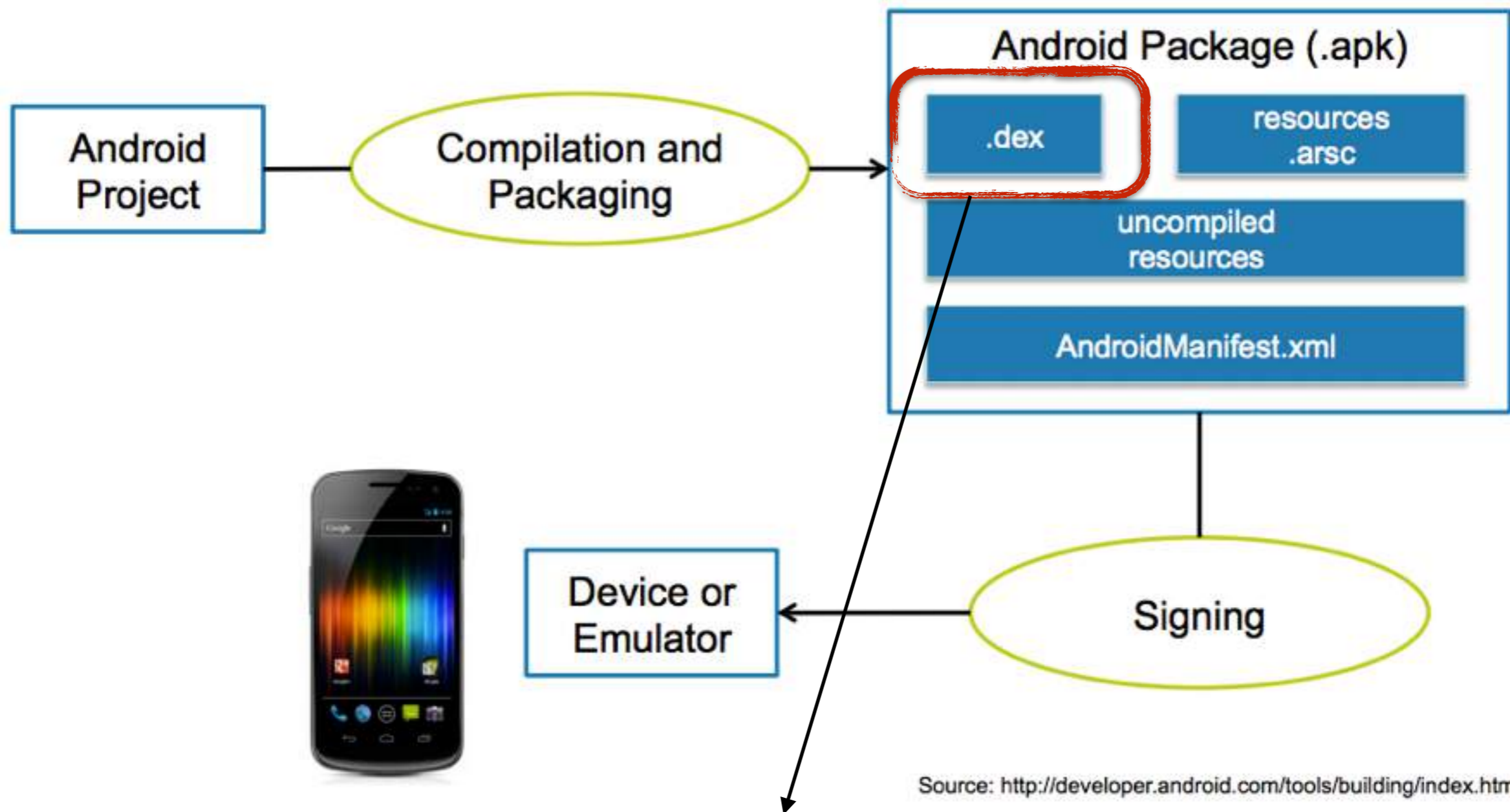
```
public void testAddNote() throws Exception {  
    solo.clickOnMenuItem("Add note");  
    //Assert that NoteEditor activity is opened  
    solo.assertCurrentActivity("Expected NoteEditor activity", "NoteEditor");  
    //In text field 0, enter Note 1  
    solo.enterText(0, "Note 1");  
    solo.goBack();  
    //Clicks on menu item  
    solo.clickOnMenuItem("Add note");  
    //In text field 0, type Note 2  
    solo.typeText(0, "Note 2");  
    //Go back to first activity  
    solo.goBack();  
    //Takes a screenshot and saves it in "/sdcard/Robotium-Screenshots/".  
    solo.takeScreenshot();  
    boolean expected = true;  
    boolean actual = solo.searchText("Note 1") && solo.searchText("Note 2");  
    //Assert that Note 1 & Note 2 are found  
    assertEquals("Note 1 and/or Note 2 are not found", expected, actual);  
}
```

UIAutomator and Espresso

- UIAutomator is another framework that allows to build tests for user apps and system apps. (integration)
 - Perfect for implementing blackbox testing techniques.
 - Provide means to inspect the layout elements in activities.
- Espresso is another framework, more suitable for implementing whitebox testing techniques (single app)

Program transformation for static analysis

Get the binary code



Source: <http://developer.android.com/tools/building/index.html>

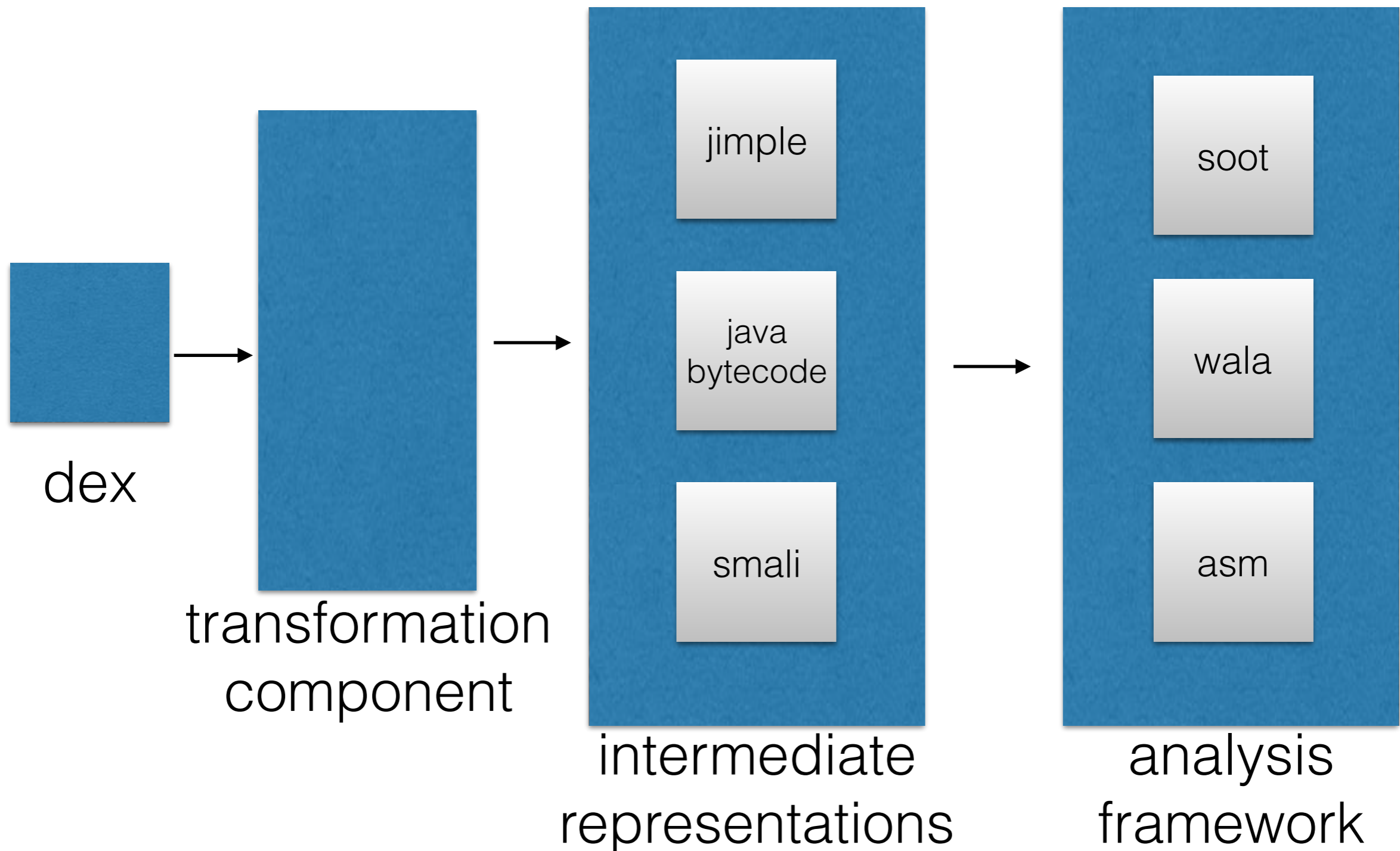
dexdump

dexdump

```
000418: 2b02 0c00 0000          |0000: packed-switch v2, 0000000c // +0000000c
00041e: 12f0                    |0003: const/4 v0, #int -1 // #ff
000420: 0f00                    |0004: return v0
000422: 1220                    |0005: const/4 v0, #int 2 // #2
000424: 28fe                    |0006: goto 0004 // -0002
000426: 1250                    |0007: const/4 v0, #int 5 // #5
000428: 28fc                    |0008: goto 0004 // -0004
00042a: 1260                    |0009: const/4 v0, #int 6 // #6
00042c: 28fa                    |000a: goto 0004 // -0006
00042e: 0000                    |000b: nop // spacer
000430: 0001 0300 faff ffff 0500 0000 0700 ... |000c: packed-switch-data (10 units)
```

not really easy to understand

Android app analysis



DEX disassemblers

- Other DEX disassembles can produce “more readable” outputs
 - **Dedexer**: turns the DEX format into an “assembly like” format. Influenced by Jasmin syntax but with Dalvik opcodes
 - **Smali/baksmali**: similar to dedexer, but well maintained (and acts as assembler as well)
 - **Androguard**: written in python. Provides some basic static analyses (check for similarities, navigate through cfgs, visualization)

Smali example

```
# class name, also determines file path when dumped
.class public Lcom/packageName/example;

# inherits from Object (could be activity, view, etc.)
# note class structure is L<class path="">;
.super Ljava/lang/Object;

# these are class instance variables
.field private someString:Ljava/lang/String;

# finals are not actually used directly, because references
# to them are replaced by the value itself
# primitive cheat sheet:
# V - void, B - byte, S - short, C - char, I - int
# J - long (uses two registers), F - float, D - double
.field public final someInt:I # the :I means integer
.field public final someBool:Z # the :Z means boolean

# Do you see how to make arrays?
.field public final someCharArray:[C
.field private someStringArray:[Ljava/lang/String;

# this is the <init> of the constructor
# it calls the <init> of it's super, which in this case
# is Ljava/lang/Object; as you can see at the top
# the parameter list reads: ZLjava/lang/String;I
# Z - boolean
# Ljava/lang/String; - java String object
# (semi-colon after non-primitive data types)
# I - integer
```

```
# these are not always present and are usually taken
# out by optimization/obfuscation but they tell us
# the names of Z, Ljava/lang/String; and I before
# when it was in Java
.parameter "someBool"
.parameter "someInt"
.parameter "exampleString"

# the .prologue and .line directives can be mostly ignored
# sometimes line numbers are useful for debugging errors
.prologue
.line 10

# p0 means parameter 0
# p0, in this case, is like "this" from a java class.
# we are calling the constructor of our mother class.
# what would p1 be?
invoke-direct {p0}, Ljava/lang/Object; -> <init> ()V

# store string in v0
const-string v0, "i will not fear. fear is the mind-killer."

# store 0xF hex value in v0 (or 15 in base 10)
# this destroys previous value string in v0
# variables do not have types they are just registers
# for storing any type of value.
# hexadecimal is base 15 is used in all machine languages
# you normally use base 10
# read up on it:
# http://en.wikipedia.org/wiki/Hexadecimal
```

Dare

- Retargeting android apps to Java bytecode
- Motivation (back in 2012): Reuse analyses that were already implemented on top of frameworks such as WALA and SOOT
- Aim: produce verifiable Java bytecode, which ensures it is analyzable by these frameworks.

Retargeting challenges

- **Type systems** are very different in DVM and JVM:
 - **Primitive assignments:** in Dalvik they specify only the width of the constant (32 vs 64 bits). No difference between float and int.
 - **Array load/store instructions:** DVM has array-specific load and store instructions for int and float arrays (a-get aput) and for long and double (aget-wide aput-wide). Type ambiguity again
 - **Object references:** Java bytecode uses null reference to detect undefined refs. Dalvik instead uses 0 to represent both number 0 and null refs.

DARE

- Works well in practice:
 - ~262,110 classes (top 50 apps of each of the 22 categories) —> successful retargeting for 99.09% of apps

Dexpler

- Converts Dalvik bytecode to Jimple intermediate representation.
- Jimple is the representation used in the Soot framework
- Built on top of dex2jars
- Uses typing inference algorithm of soot (but deals with typing ambiguities)

Jimple

```
void foo()
{
    Main this;
    double d1, d2, temp$0;
    int i1;

    this := @this: Main;
    d1 = 3.0;
    d2 = 2.0;
    temp$0 = d1 * d2;
    i1 = (int) temp$0;
    virtualinvoke this.<Main: void bar(Main,int)>(this, i1);
    return;
}
```

```
void foo() {
    double d1 = 3.0;
    double d2 = 2.0;
    int i1 = (int) (d1*d2);
    bar(this, i1);
}
```

Challenges of the Android life cycle

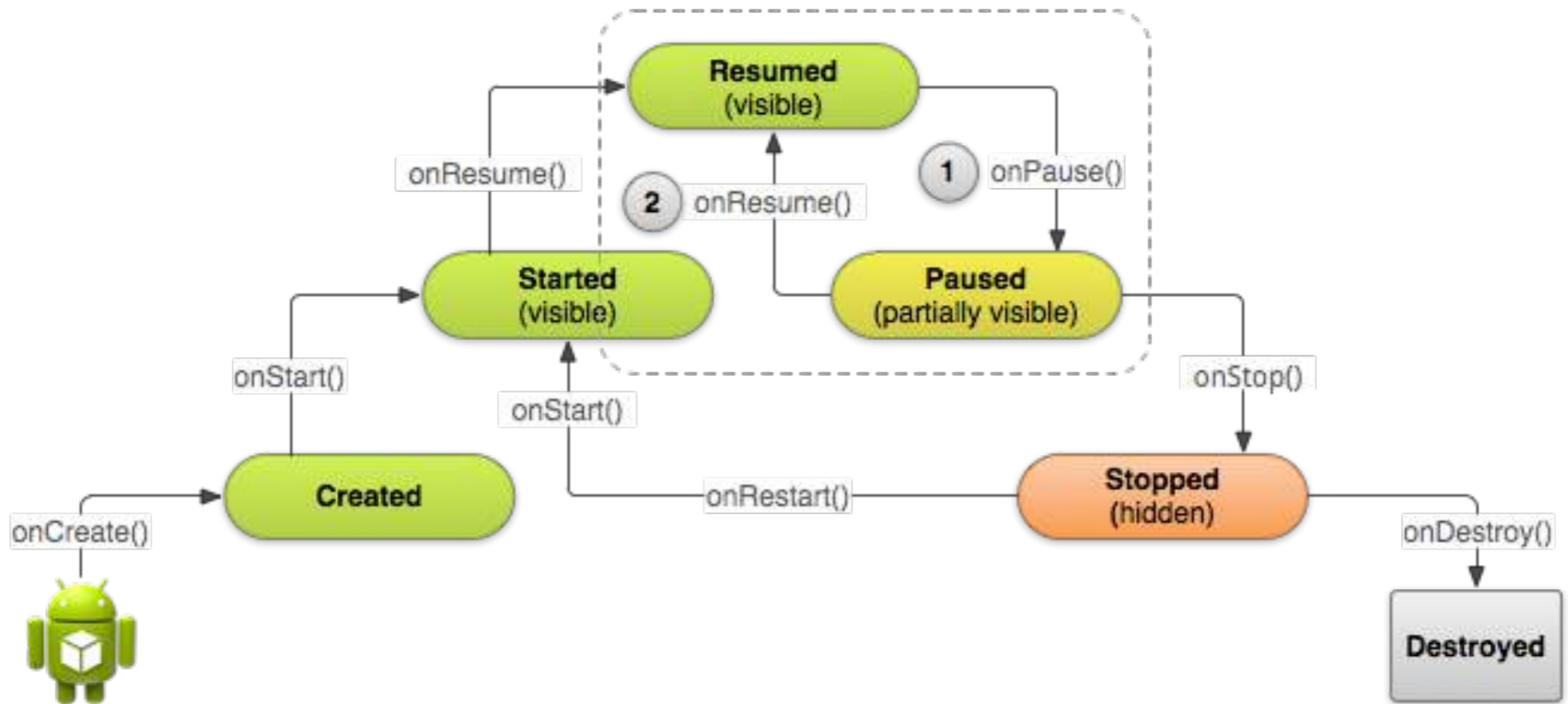
```
1 public class LeakageApp extends Activity{
2 private User user = null;
3 protected void onRestart(){
4     EditText usernameText =
5         (EditText)findViewById(R.id.username);
6     EditText passwordText =
7         (EditText)findViewById(R.id.pwdString);
8     String uname = usernameText.toString();
9     String pwd = passwordText.toString();
10    if(!uname.isEmpty() && !pwd.isEmpty())
11        this.user = new User(uname, pwd);
12 }
13 //Callback method in xml file
14 public void sendMessage(View view){
15     if(user == null) return;
16     Password pwd = user.getPwd();
17     String pwdString = pwd.getPassword();
18     String obfPwd = "";
19     //must track primitives:
20     for(char c : pwdString.toCharArray())
21         obfPwd += c + "_"; //String concat.
22
23     String message = "User: " +
24         user.getName() + " | Pwd: " + obfPwd;
25     SmsManager sms = SmsManager.getDefault();
26     sms.sendTextMessage("+44 020 7321 0905",
27         null, message, null, null);
28 }
```

read pwd from text field
when the app restarts

when the user presses a
button the pwd is sent via sms

Important to model app
life cycle and callbacks!!

Activity life cycle



Automated testing in Android

Automated Test Input Generation for Android: Are
We There Yet? — *under submission*
<http://arxiv.org/abs/1503.07217>

Fuzzing

Fuzzer



UNIX utilities



“ab'd&gfdfggg”

grep • sh • sed ...

25%–33%



Send "!o%888888888f" as command to the csh command-line shell

Invoke this with string = "%888888888f":

```
char *string = ...  
    printf(string);
```

...and made the shell hang

Fuzzing in Android

- Mildly widely used so far.
- Fuzzing mainly focused on IPC

Null intent fuzzer

- Very simple fuzzer: Null intents
 - Create null intents and see whether the broadcast receivers registered to those intents crash.

Null intent fuzzer

- **Identify targets:**
 - thanks to PackageManager
- **Generate intents**
 - Intent i = new Intent()
- **Deliver inputs**
 - sendBroadcast(i)
- **Monitor**
 - logcat.. —> NullPointerExceptions

Null intent fuzzer



“can either fuzz a single component or all components. It works well on Broadcast receivers, and average on Services”.

Only single Activities can be fuzzed.

Runs on device as an app,
opensource

Detected a serious bug in a google package that makes the phone hang

Intent fuzzer

- Works exactly like null intent fuzzer
- Static analysis component that can detect the expected structure of an intent.
- Works with inputs of primitive types

DroidFuzzer

- It focuses on generating inputs for activities that accept MIME data types (AVI, MP3, HTML files)
- It can make video player apps crash
- Tool not available

DroidFuzzer: Fuzzing the Android apps with
Intent-filter tag — MoMM 2013

Automated GUI testing in Android

Randomized GUI testing

Monkey

Tests Android apps at the GUI level
Randomly generates UI events

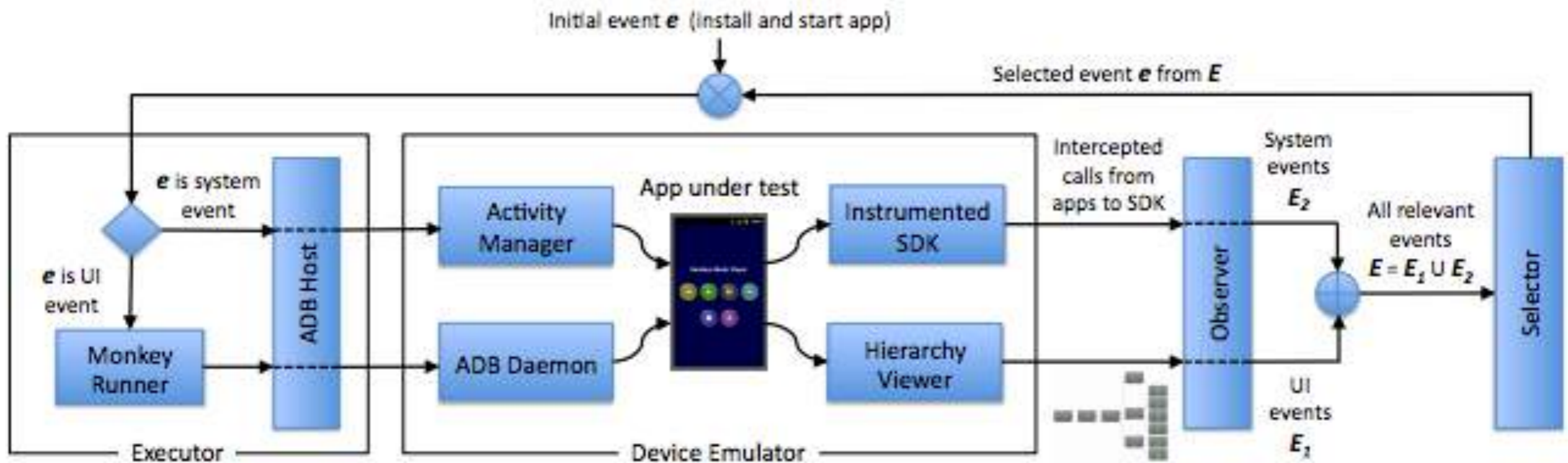
Runs on emulator or real device



```
$ adb shell monkey
```

Dynodroid

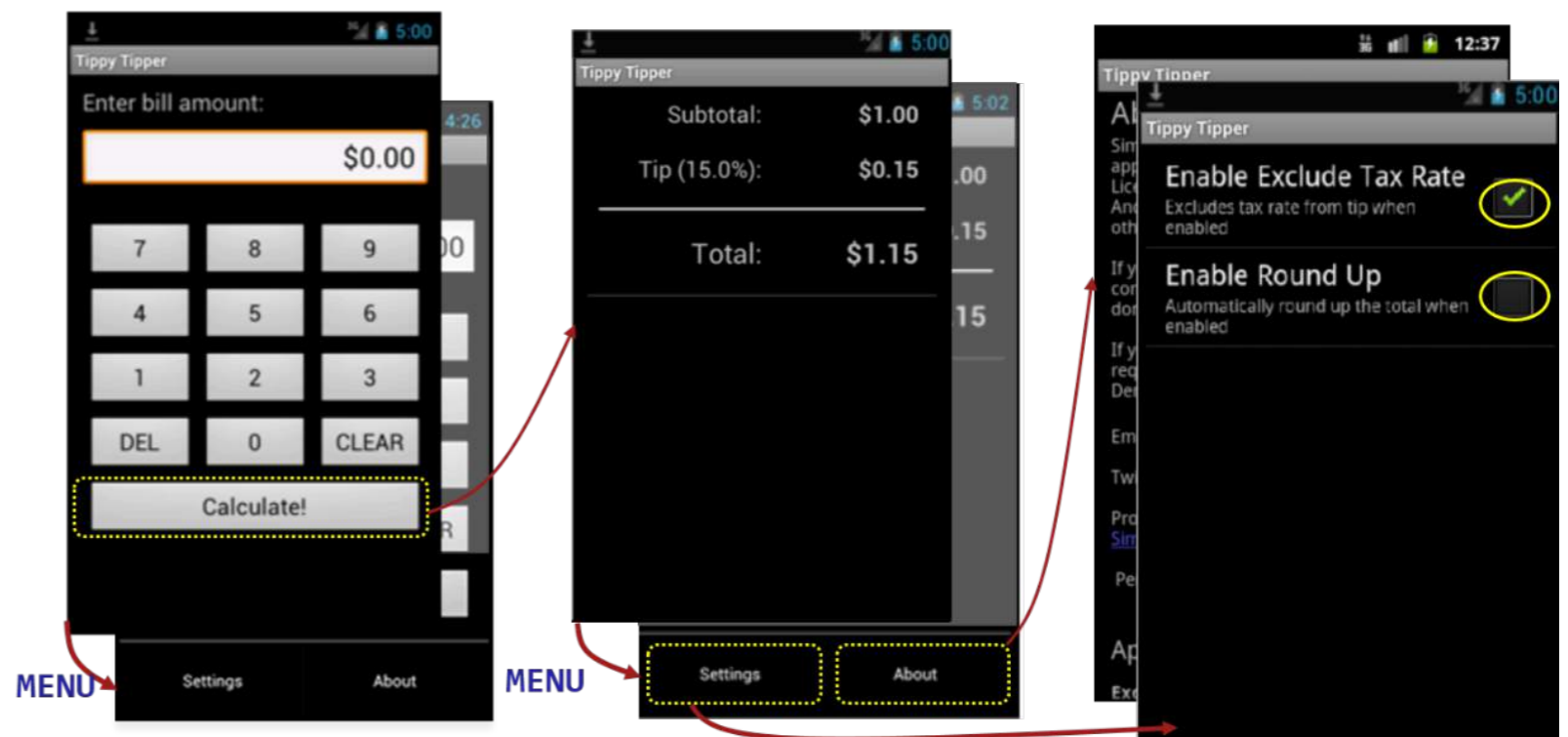
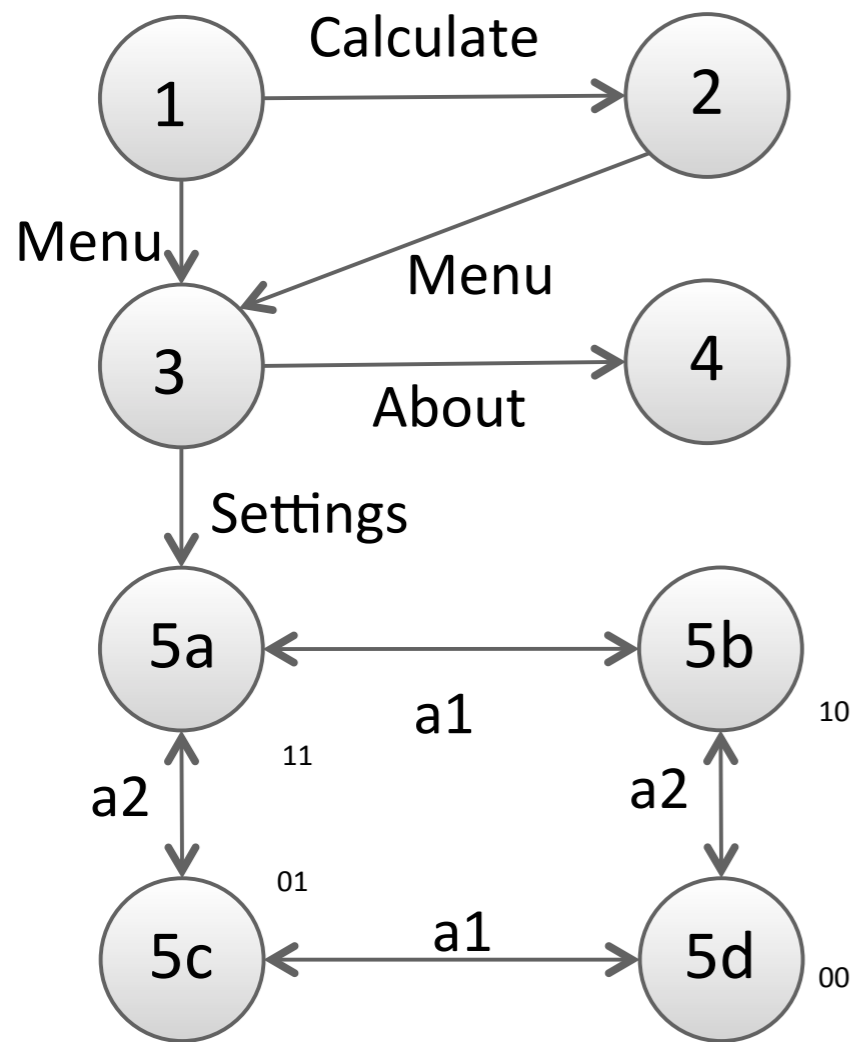
- Executor executes the event in the current state to yield a new emulator state (that overwrites the current state)
- Observer computes which events are relevant in the new state
- Selector selects one of the events to execute



Dynodroid

- How to generate relevant inputs?
 - First generate it randomly but... It lets *users* pause the automated crawling and let them provide an input.

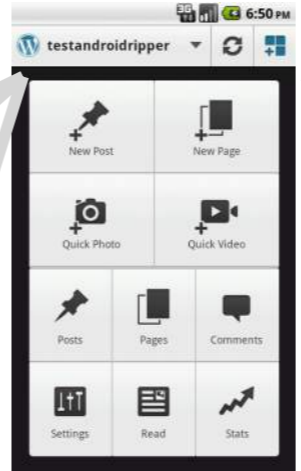
Model-based techniques



GUIRipper

- Dynamically builds FSM model
- DFS exploration strategy
- At each step it keeps list of relevant UI events

Allows users to create snapshots
and provide custom inputs



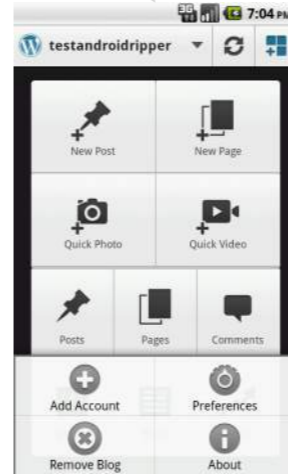
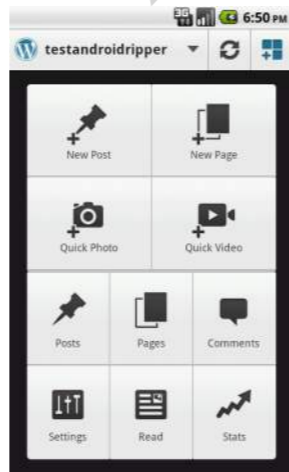
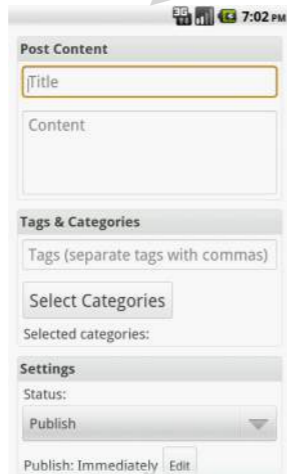
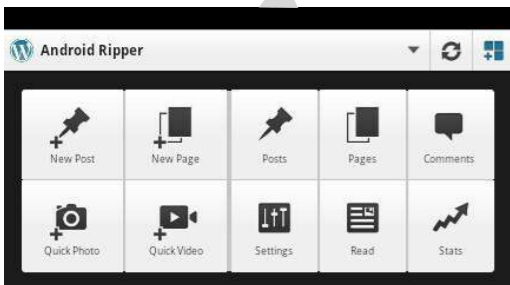
Rotate

Click New Post

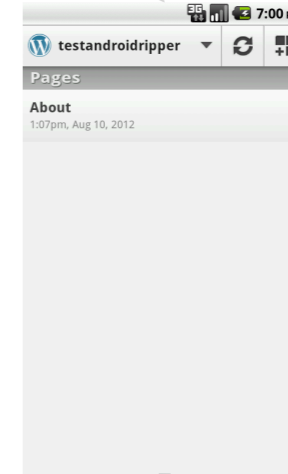
Click Refresh

Press Menu

Click Pages



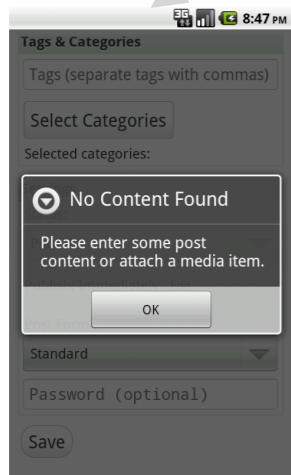
...



Click Save

Click Add Account

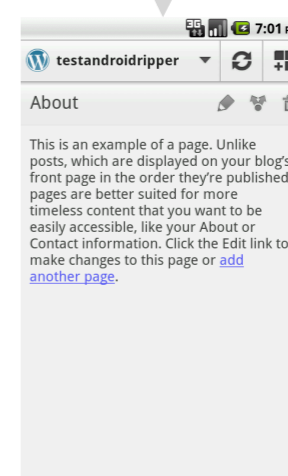
Click About



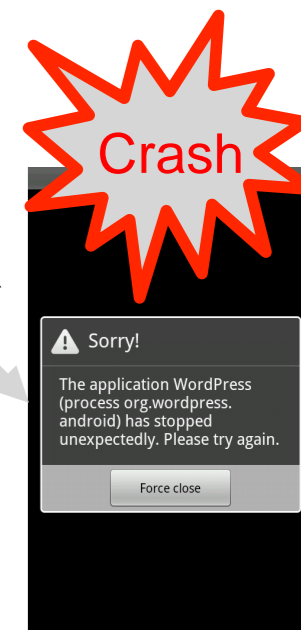
...



...



Click Edit

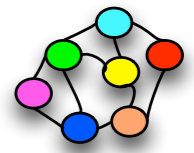


Orbit GUI testing



Android
Apps

Greybox approach



Statically extracts all the possible set of events supported by the GUI on an app.

Dynamically exercises these events on the app.

Proposed GUI model

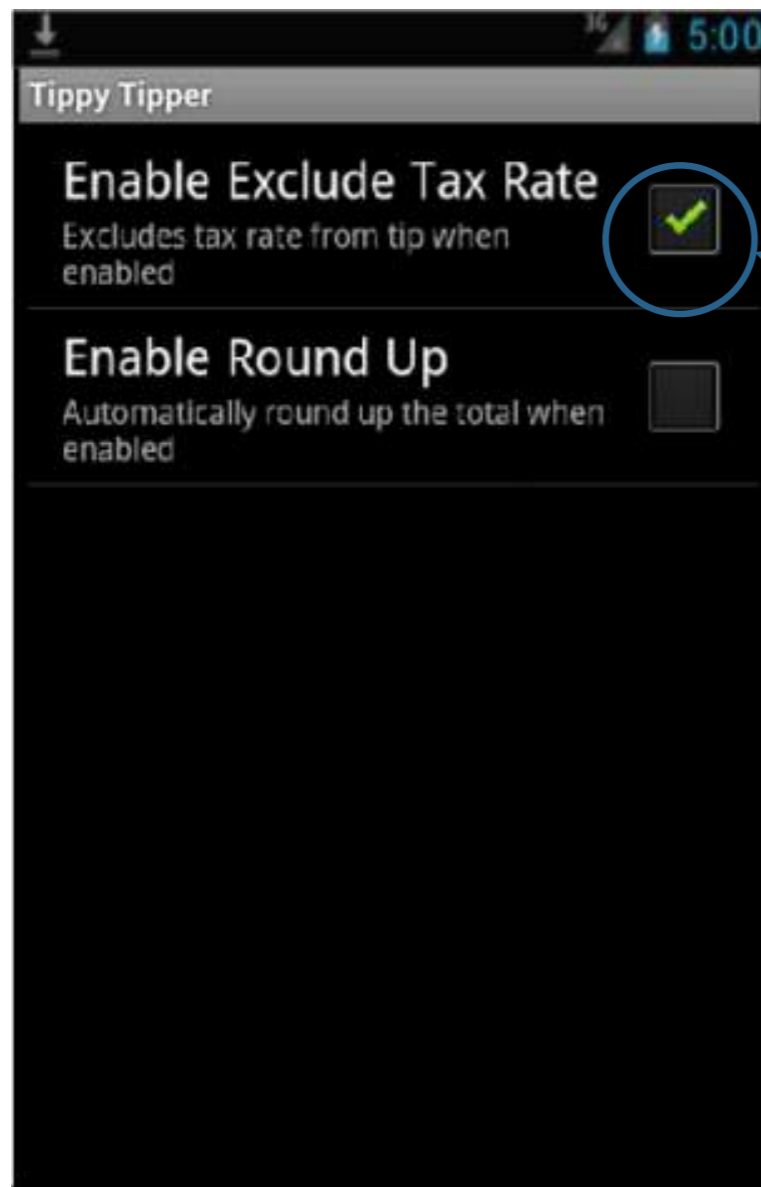
Visual Observable State

Composition of the state

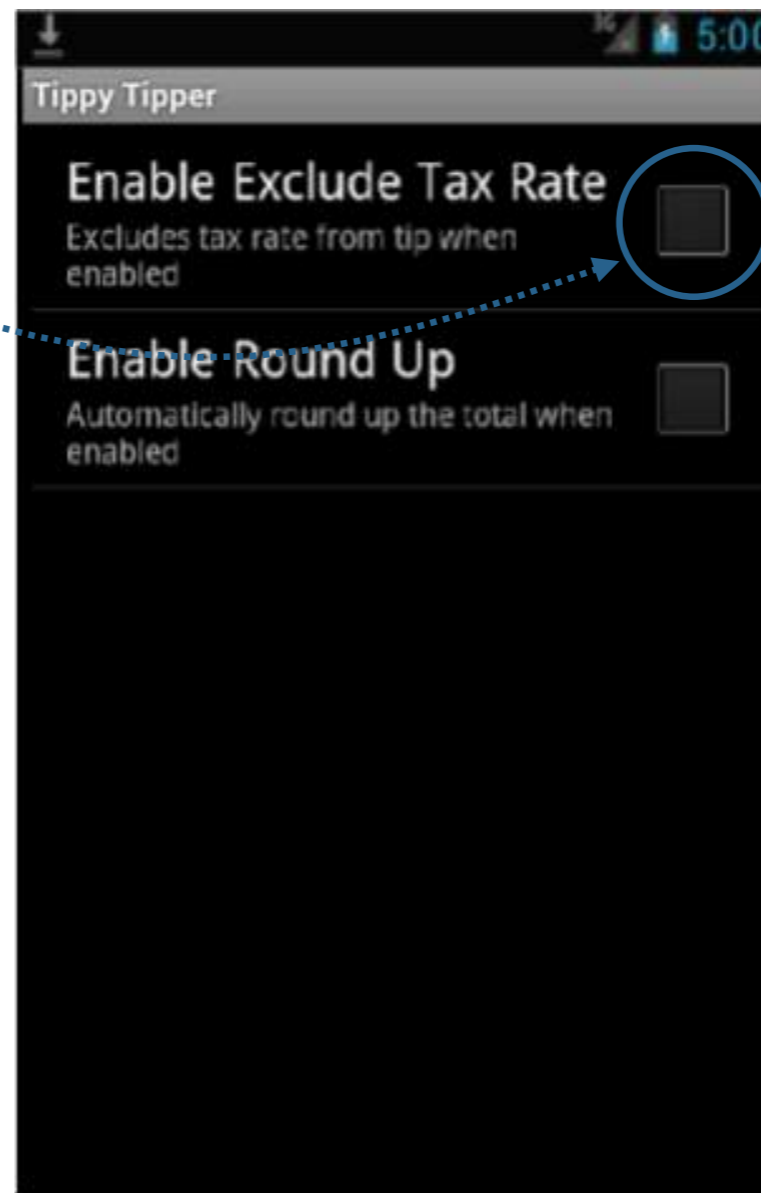
Model

A finite-state machine over visual observable states with the user actions constituting the transitions between these states

States



(5b)

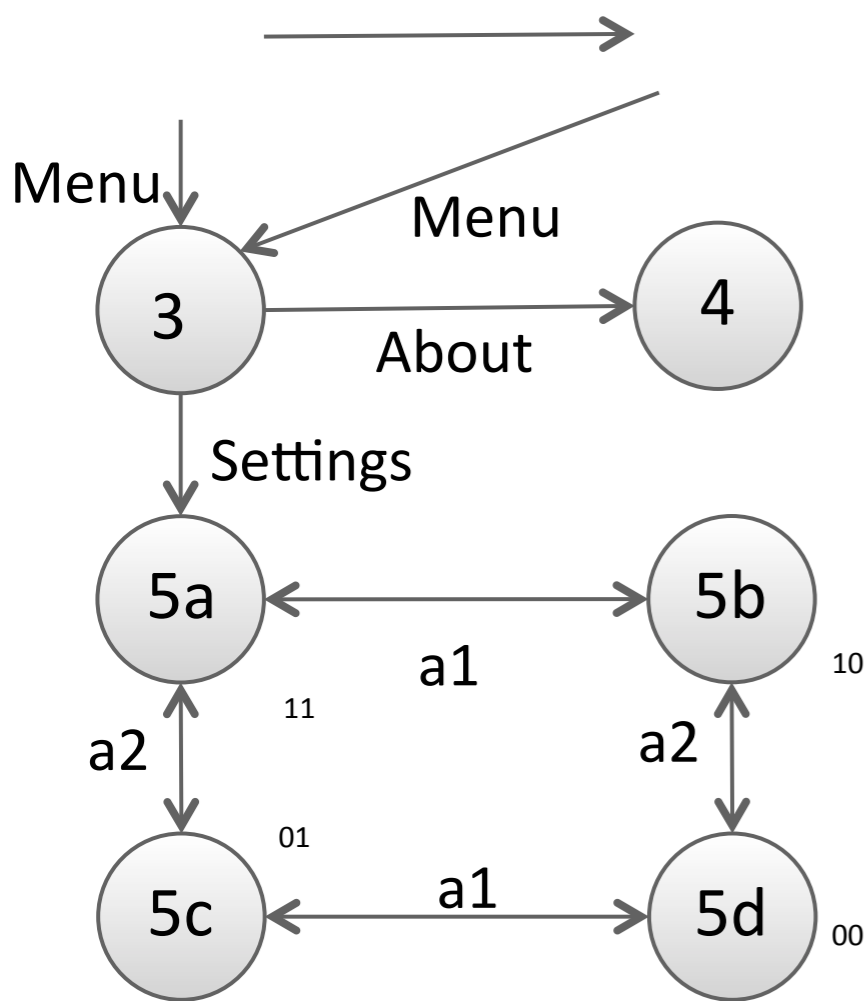


(5d)



These two states differ

Model for Simple TippyTipper



a1: Toggle exclude tax rate option.
a2: Toggle round up option.

Action Inference

TippyTipper.java

```
View btn_delete = findViewById(R.id.btn_delete);
⋮
Btn_delete.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        removeBillAmount();
        FlurryAgent.onEvent("Delete Button");
    }
});
⋮
Btn_delete.setOnLongClickListener(new OnLongClickListener() {
    public void onLongClick(View v) {
        clearBillAmount();
        return true;
    }
});
```

R.Id.java

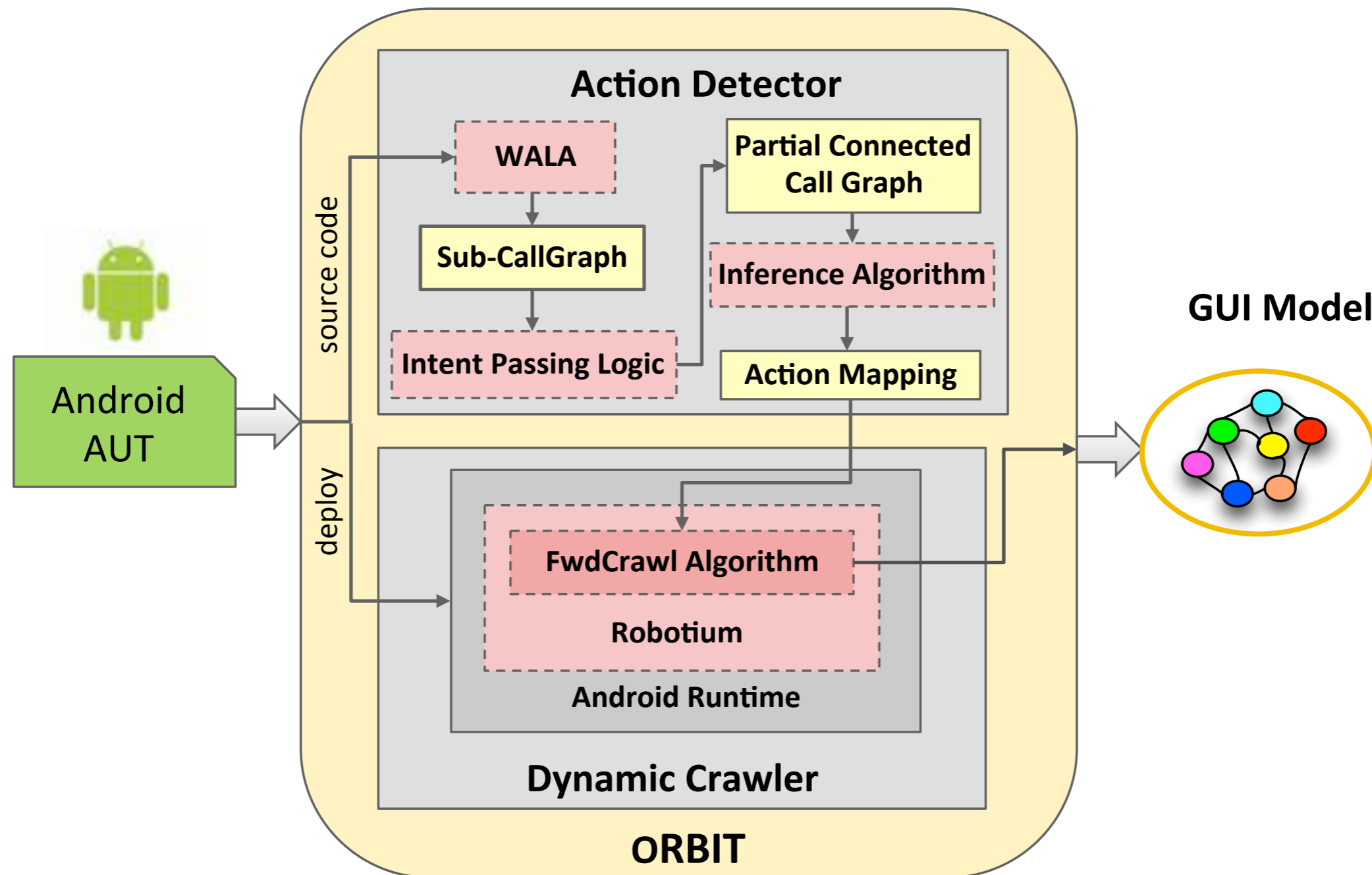
```
public static final class id {
    public static final int about=0x7f0a002d;
    public static final int btn_SplitBill=0x7f0a002b;
    public static final int btn_TipAmount1=0x7f0a0026;
    public static final int btn_TipAmount2=0x7f0a0027;
    public static final int btn_TipAmount3=0x7f0a0028;
    public static final int btn_add_person=0x7f0a001a;
    public static final int btn_clear=0x7f0a000c;
    public static final int btn_delete=0x7f0a000a;
    public static final int btn_eight=0x7f0a0002;
```

Inference: Widget *btn_delete* with *Id = 0x7f0000a* supports actions *click* and *longClick*

ORBIT: static analysis

- Identify components on which to fire an event (e.g. longClick):
 - build call graph to find methods that call `setOnLongClickListener`
 - locate statement in the caller method and get the object the listener is registered to.
 - backward analysis to get to the object initialization to get ID
 - add ID+action to list of actions to be triggered dynamically

Implementation



Automatic Android App explorer (A3E)

- Does not require access to source code
- Targeted and Depth-first visiting strategy
- Higher level of abstraction (1 activity, 1 state)
- Targeted strategy uses static analysis to compute all the activities as entry points (to analyse all of them)

Swifthand

- Dynamic model of the app. Exploration algorithm aims to reduce the number of restarts as much as possible.
- limited to touching and scrolling events

PUMA

- Framework that provides a basic monkey-like implementation.
- provides a model-based representation of an app
- possible to implement different levels of abstraction

Limitation of model-based strategy?

- Changes in internal states not represented in the model
- Problem for services

EvoDroid

- Evodroid: Uses evolutionary algorithms to guide the test-case generation towards unexplored code
- individuals as sequences of test inputs
 - mutation and crossover operators to recombine inputs
- tool not available

ACTEve

- Concolic testing tool that symbolically tracks events from their generation up to the point where they are handled in the app.
- Works both on system and UI events

JPF-Android

- extends JPF, the popular model-checking tool for Java.
- aims to explore all paths to detect deadlocks and runtime exceptions
- limitation: assumes that user provides the list of inputs.

Summary of tools

Name	Available	Instrumentation		Events		Exploration strategy	Needs source code	Testing strategy
		Platform	App	UI	System			
Monkey [10]	✓	×	×	✓	×	Random	×	Black-box
Dynodroid [11]	✓	✓	×	✓	✓	Random	×	Black-box
DroidFuzzer [12]	✓	×	×	×	×	Random	×	Black-box
IntentFuzzer [13]	✓	×	×	×	×	Random	×	White-box
Null IntentFuzzer [14]	✓	×	×	×	×	Random	×	Black-box
GUIRipper [15]	✓ ^a	×	✓	✓	×	Model-based	×	Black-box
ORBIT [16]	×	×	×	✓	×	Model-based	✓	Grey-box
A ³ E -Depth-first [17]	✓	×	✓	✓	×	Model-based	×	Black-box
SwiftHand [18]	✓	×	✓	✓	×	Model-based	×	Black-box
PUMA [19]	✓	×	✓	✓	×	Model-based	×	Black-box
A ³ E -Targeted [17]	×	×	✓	✓	×	Systematic	×	Grey-box
EvoDroid [20]	×	×	✓	✓	×	Systematic	×	White-box
ACTEve [21]	✓	✓	✓	✓	✓	Systematic	✓	White-box
JPF-Android [22]	✓	×	×	✓	×	Systematic	✓	White-box

Aim of the study



Ease of use



Android framework
compatibility



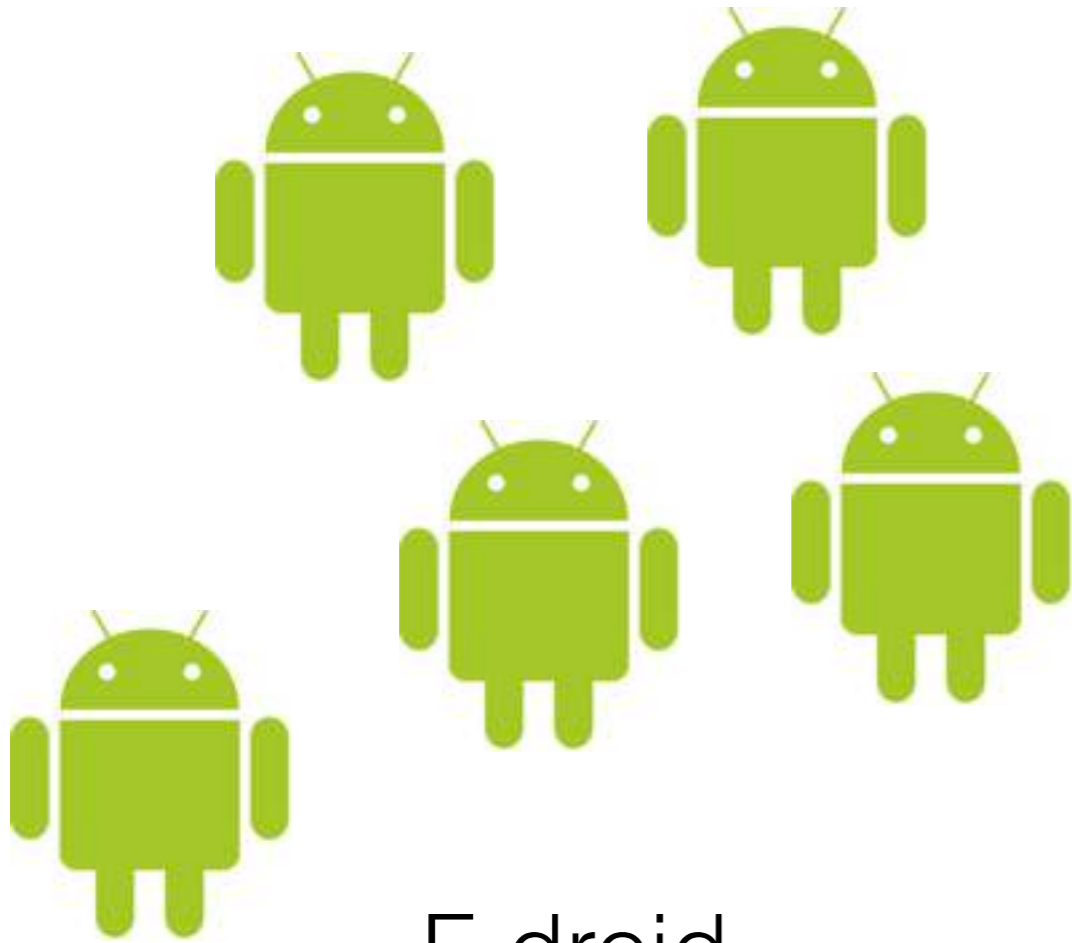
Effectiveness of
exploration strategy

Fault detection ability

Automated Test input Generation for Android:

Are we there yet? S. Roy Choudhary, A.Gorla, A.Orso - under submission

Benchmark



F-droid
68 apps

50 from Dynodroid

3 from GUIRipper

5 from ACTEve

10 from Swifthand

Logcat
Coverage

APPS



Gingerbread
(vs. 10)



Ice-cream
sandwich
(vs. 16)



Kitkat
(vs. 19)

Ubuntu

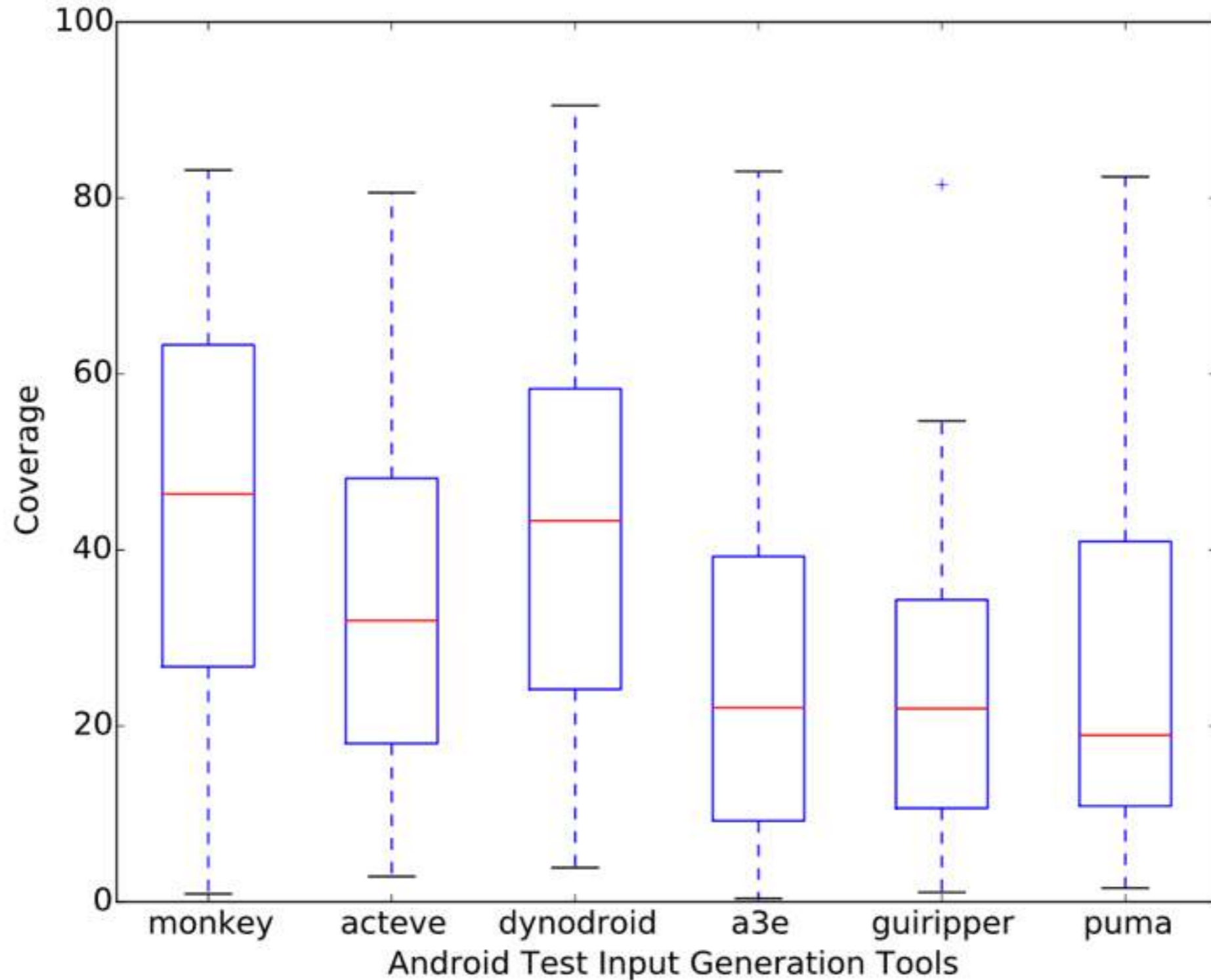


10 runs of 1 hour for each tool on each app

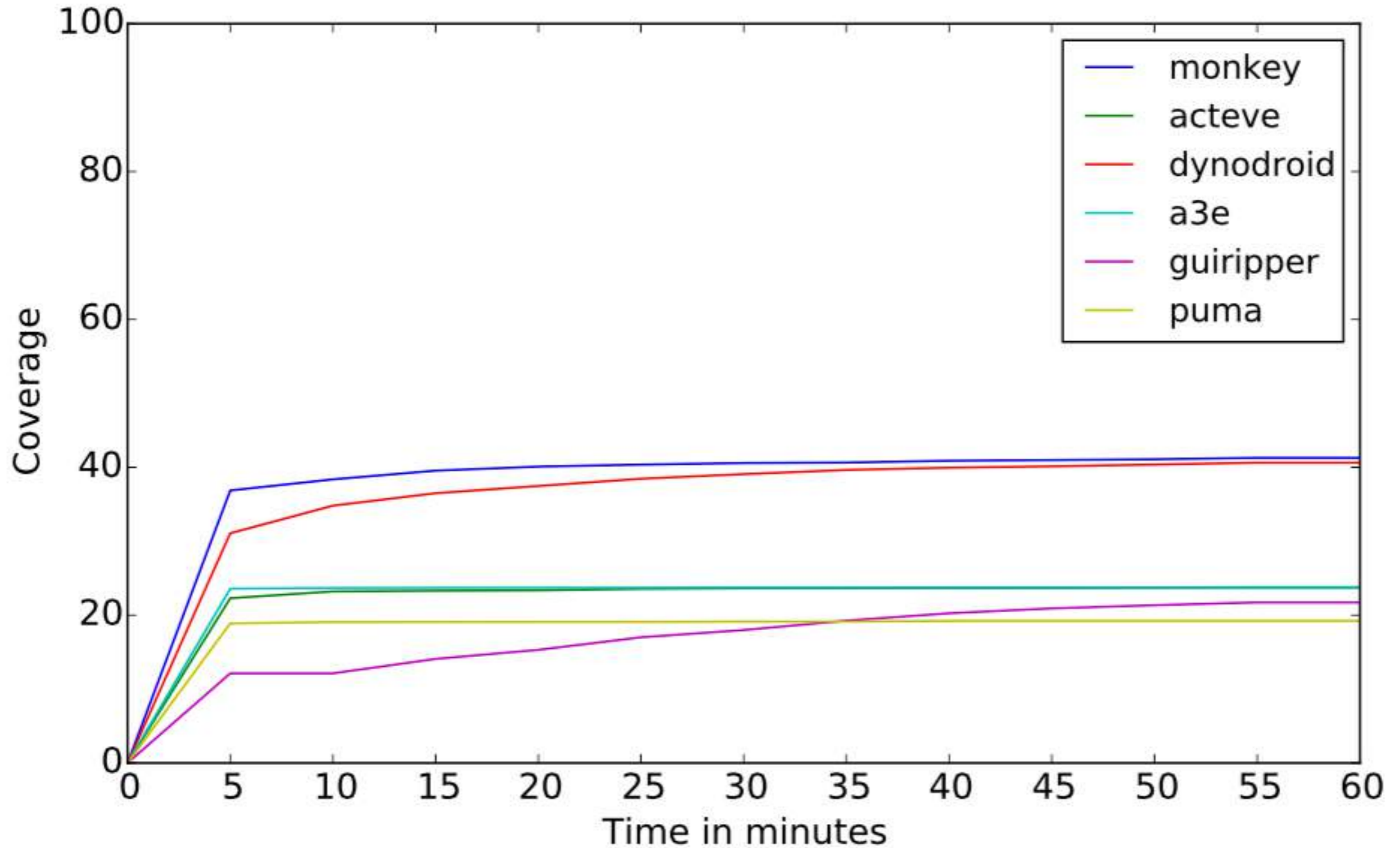
Ease of use and compatibility

Name	Ease Use	Compatibility
Monkey [10]	NO Effort	any
Dynodroid [11]	NO Effort	v.2.3
GUIRipper [15]	MAJOR Effort	any
A ³ E -Depth-first [17]	LITTLE Effort	any
SwiftHand [18]	MAJOR Effort	v.4.1+
PUMA [19]	LITTLE Effort	v.4.3+
ACTEve [21]	MAJOR Effort	v.2.3

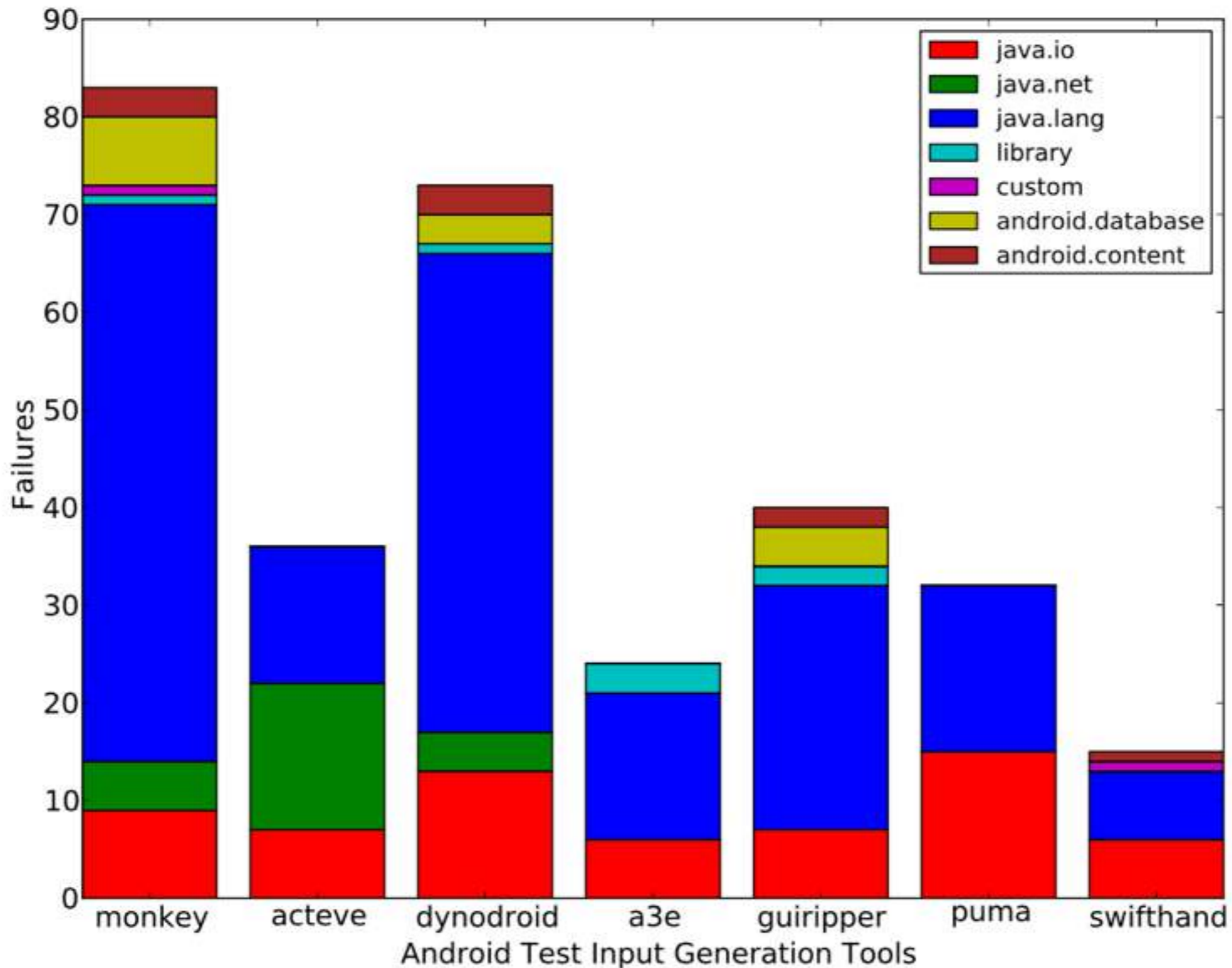
Exploration Strategy Effectiveness

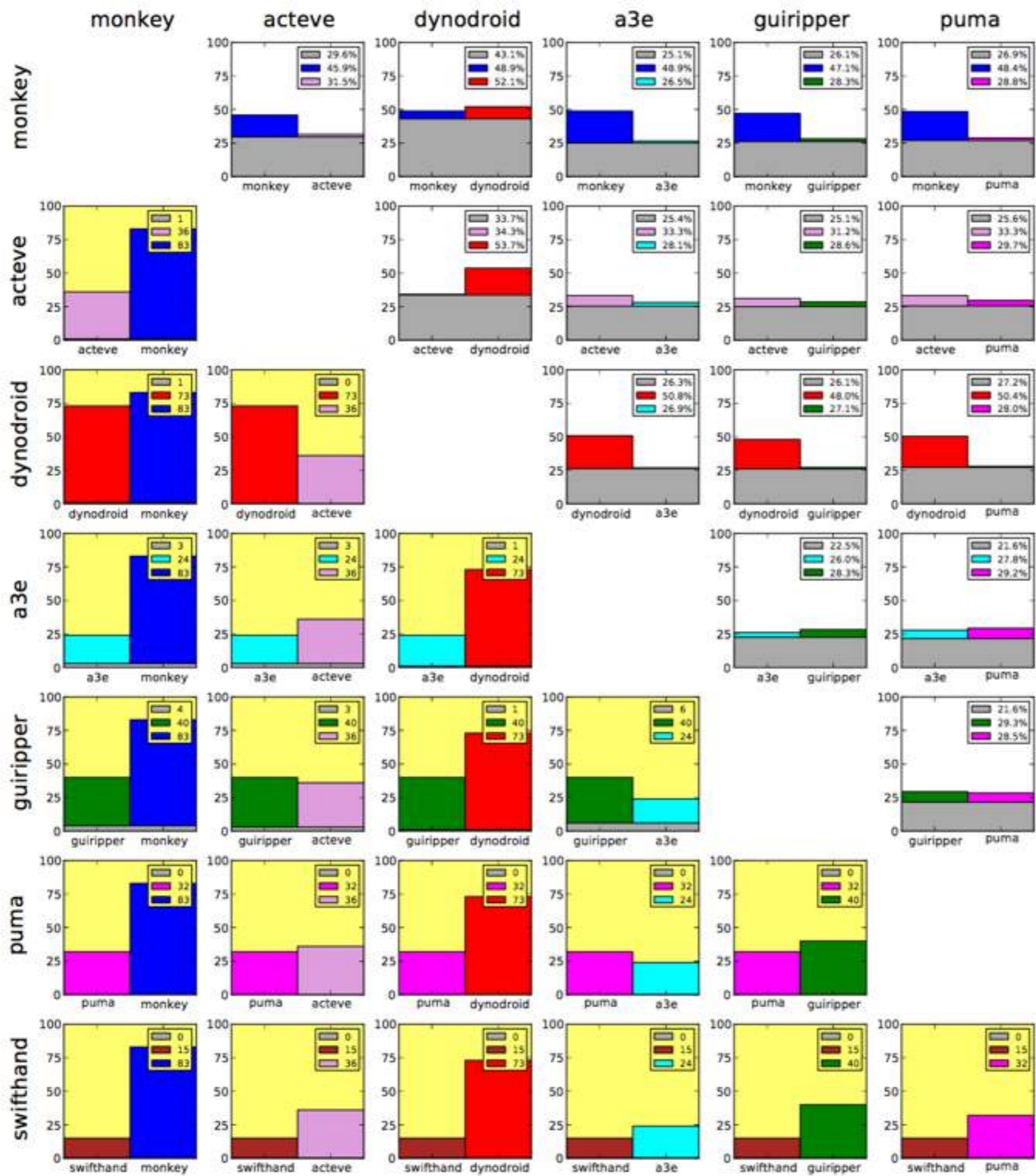


Progressive coverage



Fault detection ability





Challenges and Opportunities



System events!

- few tools support the generation of system events.
- which events to trigger and when?
- static analyses can be expensive, but may be useful to understand which events to trigger

Challenges and Opportunities

A screenshot of a form with several input fields. The fields are labeled "Line 1", "Line 2", "Town / City", "County", and "Postcode". There is a "Find Postcode" button next to the "Postcode" field. The form is enclosed in a light blue rounded rectangle.

Manually provided inputs

- Dynodroid, GUIripper only tools that consider this
- Very basic. Can we do better?

Challenges and Opportunities



exploration strategy

- e.g. Minimize restarts
- algorithm focused only on that is not enough. However, this is an interesting idea. Should be combined with other heuristics

Challenges and Opportunities



exploration strategy

- e.g. Multiple starting states
- GUIRipper can support this, but it is very basic. Has to be done manually.

Challenges and Opportunities

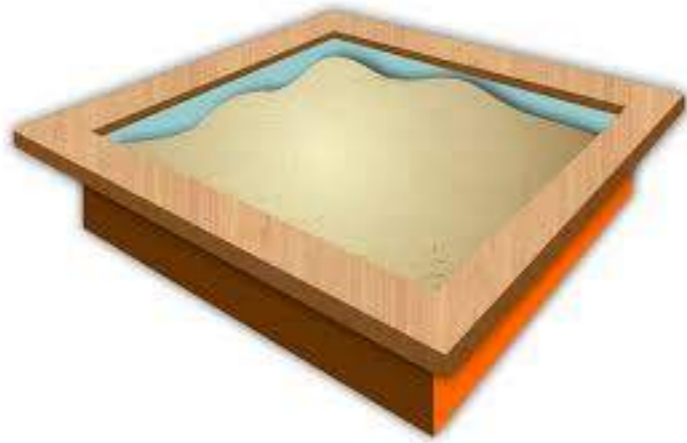


avoid side effects
across runs

- Dynodroid and A3E can clean state between runs (uninstalling app and clear data)

use our infrastructure!

Challenges and Opportunities



Sandboxing

- avoid disruptive effects of some operations

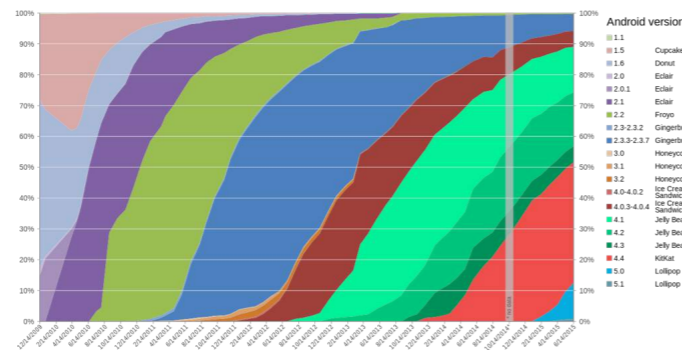
Challenges and Opportunities



Reproducible test cases

- not easy to see failure reports.
- not easy to reproduce failures.
- debugging???
- NO tool is good at this.

Challenges and Opportunities



X



Fragmentation problem

- Few commercial tools are dealing with problem
- Basic solutions (lots of manual work)