Current focus:
research and teaching in program analysis, automated software testing, security, and quality assurance for mobile applications (Android)

RESEARCH BACKGROUND

- Automated test case generation, Design for testability (with application in software recommendations systems)
- Analysis of software binaries, symbolic execution (application in vulnerability detection and crash reproduction)
- Analysis and testing of Android applications (application in automated partitioning of Android apps for enhanced security)
Mobile apps are designed for a portable device with limited resources; are instances of reactive non-terminating software with dynamic/adaptive user interfaces, and context-aware and react to changes in environment and physical factors; Apps are often hybrid -- presenting both mobile and web content.
Mobile app vs. Web app

== ‘pages’ are alike with ‘activities’
== DOM is alike with XML view hierarchy

Mobile app vs. Embedded software

== limited resources; sensors
!= mobile apps can interact with other apps

Mobile app vs. Desktop application

!= app manages it’s lifecycle
!= complex input gestures (swipes, multi-touch)
!= contextual input (location, acceleration, gyroscope, etc.)

**IS IT DIFFERENT?**

"Only about 14% of the [600 open source] apps contain test cases and only about 9% of the apps that have executable test cases have coverage above 40%.”

**OBSERVATIONS**

[P. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, ICST 2015]
USER INTERACTION TESTING

Android Instrumentation Framework

- Espresso
- JUnit
- Accessibility API

Espresso:
- Run Instrumentation and UI actions together
- Android-y matchers, synchronization between the Instrumentation and the UI thread, finds UI elements
- Android Testing Support Library merges the 3 major Google-supported Android automation frameworks:
  - basic JUnit
  - Espresso
  - UiAutomator

VERSION 2.0

EXTERNAL FRAMEWORKS

- Monkey Runner
- UI Automator
- Accessibility API

VERSION 1.0

- Monkey
- UI Automator
- Accessibility API

VERSION 1.0:
- Acceptance tests for Android and iOS, translates Cucumber into Robotium
- Hybrid apps: finds UI components
- Selenium-style with webdriver, clients in different languages

VERSION 2.0:
- TouchUtils
- Animations
- Sync/Wait

EXTERNAL FRAMEWORKS:
- Robotium
- Calabash
- Appium
- Selendroid
- Appium

- Android-y matchers, synchronization between the Instrumentation and the UI thread, finds UI elements

Kostiantin Rubinev
Politecnico di Milano
STATE OF THE AFFAIRS

• Overlapping functionality between different frameworks
• Poor and conflicting testing documentation (esp. Google)
• Game app testing is weak/missing. Game apps bypass Android Views to draw and thus cannot be tested as normal view resources (Testdroid solution has approached it through image recognition)
• Activity testing is slow, requires mocking, and has to run on Emulator/Device (addressed by http://robolectric.org - mimic how Android creates Activities and drives them through their lifecycle)
• Active improvements in 2014-2015
• Automated test execution, but little or no automated test case generation

ANDROID INFRASTRUCTURE

RESEARCH SAMPLES

CHALLENGES

Dynodroid: An Input Generation System for Android Apps
Aravind Machiry Rohan Tahiliani Mayur Naik

ABSTRACT
We set out to build a system for generating inputs to mobile apps on Android, the dominant mobile app platform. Unlike traditional software testing environments, mobile apps run on highly heterogeneous devices and their functionalities vary greatly. Moreover, unlike most software, mobile apps run on a mobile operating system, which adds an extra level of complexity. We wanted to develop an input generation system that can systematically generate inputs to mobile apps, and to that end we developed Dynodroid. Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets. We evaluated Dynodroid on 50 open-source Android apps, and compared Dynodroid to humans, who are better at providing intelligent inputs, and Monkey, which is the testing framework included in the Android SDK. Dynodroid also found 9 bugs in 7 of the 50 apps, and 6 bugs in 49 of the 50 apps. Monkey and Dynodroid covered 45% and 50% of each app's Java source code on average, while humans covered 55% of each app's Java source code on average. Monokey, humans, and Monkey covered 55%, 60%, and 53%, respectively, of each app's Java source code on average. Dynodroid, humans, and Monkey covered 55%, 60%, and 53%, respectively, of each app's Java source code on average. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to execute a game. For this reason, Dynodroid allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks. Dynodroid also allows a user to replay a game. Dynodroid also finds bugs such as memory leaks.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.

Dynodroid is an event-driven program that interacts with its environment—simulating an end user in a manner that allows Dynodroid to exercise app functionality and observe its reactions. Dynodroid exercises the selected widget, and identifies dynamically which events are relevant to the app in the current state, then observes how the app reacts to those events. This cycle, in which Dynodroid observes, selects, executes, and observes again, is repeated for each selected widget until Dynodroid has observed all widgets.
Overview

• Finds relevant system events: what app can react to at each moment in execution
• Considers both UI (leaf/visible nodes on View Hierarchy) and System events (broadcast and system services)
• Randomized exploration (select a widget by penalize frequently selected ones)
• The approach is black box, it works iteratively and finds registered listeners dynamically
• Allows intermediate manual input

\textbf{DYNODROID}

Evaluation

• Dynodroid exclusively covers 0-26\% of code; an average of 4\%
• Dynodroid + Manual input covers 4-91\% of code per app; an average of 51\%

Approach

• Test adverse conditions, yet injecting expected events
• Injecting neutral system events (An event sequence \( n \) is neutral if injecting \( n \) during a test \( t \) is not expected to affect the outcome of \( t \))
• Examples: Pause \( \rightarrow \) Resume; Pause \( \rightarrow \) Stop \( \rightarrow \) Restart; Audio focus loss \( \rightarrow \) Audio focus gain;
• Orig. test cases are redundant. Optimization: omit injecting \( n \) in abstract state \( s \) after event \( e \), if \( (n, s, e) \) already appears in the cache (uses View Hierarchy)

\textbf{THOR}
Overview

• Systematic system event fuzzing based on existing test cases with the focus on activity lifecycle changes
• Finds suitable locations for injecting events in TCs
• Localizes faults (a variant of delta debugging for failing TCs)
• Minimizes rerunning (ignores injections that are redundant)
• Provides fault classification and criticality (Element disappears; Not persisted; User setting lost; Crash; Silent fail; Unexpected screen; etc.)

Evaluation

• Works for Robotium (and Espresso) test suites
• 4 open-source Android apps (with a total of 507 tests)
• 429 tests of a total of 507 fail in adverse conditions
• Revealed 66 distinct problems
• 18 of the 22 critical bugs found by Thor are not crashes

WHAT'S NEXT?

• Activities in isolation - business logic, unit testing
• Message passing between activities (Intents), integration testing
• Explore application GUI; guided/random exploration;
• Activity/Fragment/app Life-cycle changes/related interactions
• Interaction with OS, sensors
• Interactions with other apps and services, web info
• Security/Privacy/Energy testing (not covered here)
• Distributed testing (run on multitude of read devices and simulators)

WHAT TO TEST AND HOW?
CHALLENGES

- Android emulators are slow and unstable
- "Flaky test" issue
- Input generation (Input data like user account is impossible to generate automatically)
- Supporting a wide range of devices, platforms and versions
- GUI models are limited ("Some events may change the internal state of the app without affecting the GUI") yet allow to cover large parts of app behavior
- Isolating app behavior yet testing platform specific functionality
- State-sensitivity and state explosion

Developer is not provided with tools or models to manage the state configuration during design time, while testing approaches seek to explore the visible state and basic system interactions.

CURRENT WORK @ POLITECNICO

COMPREHENSIVE EVENT-BASED TESTING

- Static analysis for: resource release, best practices, double instantiation (e.g., location acquire/release)
- Framework for lifecycle testing, works with Espresso/Robotium (e.g., test app while after Activity.onCreate())
- Temporal assertion language for event-based testing works with Espresso/Robotium built on RxJava (express causality and order)

**RESEARCH PROPOSALS**

**Static analysis:** integrating with automated program repair; novel dynamic/adaptive interface checks.

**Temporal assertion generation:** automated assertion placement; automated collection of oracles for temporal assertions; automated test case generation.
RESEARCH PROPOSALS

Layout/fragmentation issues:

• automated testing dynamic/adaptive interfaces;
• automated generation of layout oracles and constraints;
• optimal device selection for dynamic interface testing.

Konstantin Rubinov
Politecnico di Milano

find me:
http://futurezoom.in
email: konstantin.rubinov@polimi.it