Automated extraction of augmented models for native and hybrid mobile applications in Android

Santiago Liñán Romero

December, 2018
Version: Full draft - Prior Advisor review

Bogotá, Colombia

Systems and Computing Engineering Department
Faculty of Engineering
The Software Design Lab

Thesis submitted to the Faculty of Engineering in partial fulfillment of the requirements for the degree of MSc. in Software Engineering

Automated extraction of augmented models for native and hybrid mobile applications in Android

Santiago Liñán Romero

Advisor Ph.D. Mario Linares Vásquez
Systems and Computing Engineering Department
Universidad de los Andes, Bogotá, Colombia

1. Reviewer Ph.D. Nicolás Cardozo
Systems and Computing Engineering Department
Universidad de los Andes, Bogotá, Colombia

2. Reviewer Ph.D. Gabriele Bavota
Faculty of Informatics
Università della Svizzera italiana (USI), Lugano

December, 2018

Santiago Liñán Romero

Automated extraction of augmented models for native and hybrid mobile applications in Android

Thesis submitted to the Faculty of Engineering in partial fulfillment of the requirements for the degree of MSc. in Software Engineering, December, 2018

Reviewers: Ph.D. Nicolás Cardozo and Ph.D. Gabriele Bavota

Advisor: Ph.D. Mario Linares Vásquez

Universidad de los Andes

The Software Design Lab

Faculty of Engineering

Systems and Computing Engineering Department

Cra 1 # 18A - 12

111711 and Bogotá, Colombia

Abstract

Mobile software development involves significant challenges to developers such as device fragmentation (*i.e.,* enormous hardware and software diversity), event-driven programming (*i.e.,* programming based on user interactions, sensor readings and other events where the program must react) and continuous evolving platforms (*i.e.,* fast changing mobile frameworks and technologies). This can lead programmers to error-prone code, because of the multiple combinations of external variables that must be taken into account in an app development process. Thus, testing is an underlying necessity in mobile applications to deliver high quality apps. However, defining tests suites for app development is a difficult task that requires a lot of effort, because it must consider all the possible states of an app, its context (*e.g.,* device in which is running, sensors, touch gestures, screen proportions, connectivity), the technologies involved in the development of the app (*e.g.,* native, native written in Javascript, hybrid) and a large combination of mobile devices and operating systems.

Previous efforts have been done to extract models that support automated testing. However, as of today there is not a single model that synthesizes different aspects in mobile apps such as domain, usage, context and GUI-related information. These aspects represent complementary information that can be mixed into a single and enriched model. In this paper, we propose a multi-model representation that combines information extracted statically and dynamically from Android apps. Our approach allows practitioners to automatically extract augmented models that combine different types of information, and could help them during comprehension and testing tasks.

Contents

1 Introduction

.

> *The most exciting mobile trend is full Qwerty keyboards. I'm sorry, it really is. I'm not making this up*
>
> **— Mihal "Mike" Lazaridis on iPhone, May 2008**
> (Founder of BlackBerry)

Modern mobile application testing is becoming more complex than ever before due to fragmentation and context-aware software. The majority of the mobile applications that we use daily (*e.g.,* Facebook, Waze, Uber, Web browser) rely on external web-services, connectivity methods (*e.g.,* Wi-Fi, Bluetooth, celullar networks) and sensors (*e.g.,* GPS, proximity sensor, cameras) that interact together. This amalgam of technologies increases the number of states and contexts to be considered when testing applications. Additionally to the previous examples, there is a growing number of IoT apps that have continuous interactions with their context (*e.g.,* health care applications in hospitals [39], tourist attractions [6]).

Context-awareness in mobile applications is fundamental to create robust tests. "Since 1991, context-awareness computing has been established as a well known research area in computer science" [38]. The definition has evolved from desktop applications to mobile applications. Regarding apps, context includes many more external variables (e.g., device hardware, device connectivity, battery charge, Android version, external sensors inputs, time).

Moreover, devices where mobile applications run are becoming more powerful and capable to capture, process and transmit information of their context. To capture data, the Android Platform supports more than 12 different sensors (*e.g.,* accelerometer, gravity, light, magnetic field, pressure, temperature)[14]. Besides multiple sources of data, there are numerous possibilities concerned to connectivity.

[1] **Bibliographical note:** Part of this work has been published as a NIER conference paper in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). Automated Extraction of Augmented Models for Android Apps[33]

An Android phone can interact with another device via Bluetooth, NFC, Wi-Fi P2P, USB, and SIP. [11]

As a consequence of the complexity of an app and its context, testing is a hard labour for developers. Context changes can induce unforeseen faults and errors that traditional black-box or grey-box techniques do not take into account. Let us use a fictional app as an example to illustrate the case: A delivery driver is using an app that indicates her the route to deliver her packages; the battery of her smart-phone is draining fast, then, in order to extend battery life the device turns off background location and network services. The delivery driver thinks that she has finished her tasks because the application stopped showing her directions. However, the app crashed as a consequence of an unforeseen change in the device connectivity, due to battery saving configurations in the device.

In light of the time and effort required to take into account all the possible scenarios and generate the corresponding test cases, automation of testing processes become an essential job during app development, however, the most part of mobile testing is done manually by developers owing to several factors, including that current tools do not provide testers with testing capabilities for complex interactions and contextual events [21], [31], [23], [30].

1.1 Thesis goals

To main goal with this thesis is to *develop an approach for improving mobile software testing, automating multi-model generation, app exploration and crash detection for native and hybrid apps.* In particular, the specific goals are:

- Define an approach for improving automated testing of mobile apps, by taking advantage of programmatic extraction of context, usage, domain and GUI models from an Android application.

- Develop a tool that automatically (i) extract multi-models, and (ii) performs rip-based crashes detection.

- Evaluate the performance of the proposed tool (in terms of GUI exploration and crashes detection) when compared to state-of-the-art rippers.

1.2 Thesis contributions

We propose an approach for improving automated testing, by taking advantage of programmatic extraction of context, usage, domain and GUI models from an Android application. Along with the automated extraction, we propose the conception of a multi-model (or augmented model) that combines the aforementioned models and can be used to support testing related tasks such as test cases generation, execution, and documentation.

Manual creation of those models is time consuming; that is the reason why extracting models programmatically from static code analysis and dynamic exploration is ideal. Although models by their own are useful artifacts to document the application, we propose the generation of an augmented model mixing key information of different models. The augmented model we propose synthesizes aspects from the graphical user interface, domain and context. This multi-model has new information that only can be obtained by the intersection of the aforementioned individual models. That said, multi-models lay the foundations for future efforts to improve model-based testing in mobile platforms based on richer information contained in augmented models.

Throughout our experiments of automated extraction of models and exploration of Android apps, we found a significant amount of hybrid apps: applications that contain an embedded web browser in which all visual and logic components are contained. These are HTML, JS and CSS web applications that do not make use of the Android UI framework. Because of this singularity, we had to implement a technique to obtain the graphical hierarchy contained in applications of this kind.

We validated the extraction of multi-models exploring native open source Android apps. The resulting augmented models contain a significant number of states that were found due to induced contextual changes in the devices. In average, applications contain 6.33 states associated to contextual events. Provided this, we confirmed that each of the models included in the proposal is relevant and have an impact on the behavior of the applications.

Additionally, we conducted an experiment with 20 hybrid applications available in the Google Play Store and 15 open-source native applications found in F-Droid, and tested our GUI ripping technique. We found that our approach outperforms Google's testing tools such as Firebase Test Lab [12] and Android UI/Application Exerciser Monkey [15] in hybrid apps exploration. Our technique explores hybrid application with less than 5% of the states wrongly classified, compared to 44% of other rippers which are not oriented to hybrid applications exploration.

Our approach enabled us to find crashes in 40% of the hybrid applications tested, whereas Firebase Test Lab [12] and Monkey [15] found bugs in just one of the hybrid applications.

1.3 Thesis Structure

Chapter 2 presents related work respecting multi-model testing, frameworks, GUI ripping tools and hybrid applications. **Chapter 3** describes the proposed approach describing carefully every detail of **RIP** (our ripper), its technology and implementation. **Chapter 4** introduces an empirical study, based on three research questions answered with the aid of experiments and case studies. **Chapter 5** concludes the document, revising our contributions and defining future work.

2 Related Work

> *An iPod, a phone, an internet mobile communicator... these are NOT three separate devices! And we are calling it iPhone! Today Apple is going to reinvent the phone. And here it is.*
>
> **— Steve Jobs**
> MacWorld 2007

2.1 Native applications

Native apps are applications built for particular platforms. These applications are written in Java, Dart or Kotlin for Android, and Swift or Objective-C for iOS. Regarding native applications, automated mobile app testing has been explored profusely in many areas (*e.g.,* frameworks, automation APIs, testing techniques, GUI exploration, security, usability, error reporting tools)[30], [46], [7], [22]. In this section we focus specifically on two aspects: testing frameworks and GUI ripping tools (due to the nature of model extraction based on GUI events), which are closer to our proposal.

2.1.1 Testing frameworks

Frameworks for mobile testing define principles, concepts and architectures to structure testing processes. To this end, one of the frameworks where automated extraction of augmented models have been explicitly defined is CEL [30]. CEL is based on three main principles: *continuous*, *evolutionary* and *large-scale*. The first principle, *continuous*, refers to the fact that mobile apps should be tested under multiple environmental conditions and according to different goals. The second principle, *evolutionary*, sets forth that testing artifacts like the models, should adapt to changes in source code, environment, and in-the-wild usages. Lastly, the *large-scale* principle proposes an engine to execute test cases in real and emulated devices to tackle fragmentation.

With this in mind, the CEL framework argues that there must be a "models generator" component that combines multiple models such as GUI, usage, and contextual models to finally create a multi-model representation of an app under test. This multi-model can be used for the evolutionary generation of testing artifacts [30]. CEL affirms that current approaches for deriving representations of apps are severely lacking a multi-model-based approach that might significantly improve the utility of model-based testing [30]. Having said that, our approach of extraction of an augmented model fits CEL's principles and architecture, enhancing GUI exploration with complementary models. Note that the CEL paper does not provide any detail regarding how a multi-model should be; thus, we are the first to instantiate the multi-model concept as proposed by CEL.

2.1.2 GUI ripping tools

GUI ripping tools simulate real user events on an Android device to explore an application GUI. The majority of these tools detect and report crashes generated during the exploration. Coupled with detection of crashes, others of these tools also reconstruct GUI models resulting from the exploration (*e.g.,* MonkeyLab [32], AimDroid[18], Android Ripper [3]). Previous effort has been also focused on tools that build testing suites based on their exploration strategy (*e.g.,* Sapienz [34], MobiGUITAR [2]).

DroidBot[27] is another tool that uses a model-based strategy to automatically explore mobile GUI based on a DFS effective strategy. It generates inputs and a transition model between different states of the application. One of the main features of DroidBot is that it does not require app instrumentation and is mean to run in almost any Android device.

Firebase Test Lab [12] is one of the industry leaders in mobile testing. It has a cloud-based app-testing infrastructure that enables concurrent execution of tests with and without instrumentation. Firebase Test Lab Robo Test is one of their services: 'Robo test analyzes the structure of your app's UI and then explores it methodically, automatically simulating user activities' [12]. It cloud service allows testers to run Robo test on virtual and physical devices in parallel, detecting crashes and performance issues.

The aforementioned tools have advanced significantly in terms of algorithms to explore apps' GUI. They systematically create state diagrams based on GUI information and GUI states exploration. However, GUI ripping tools lack information about the execution context that could help to determine contextual states, *e.g.,* a navigation app without GPS and Internet can not be as functional as it was intended to be,

because turning GPS on and off in the same activity could result into two totally different states. GUI rippers also lack information about domain and usage of the application. However, effective mobile testing requires considering different types of information (*i.e.,* GUI, domain entities, contextual states, real usages) because combining more information could drive to exploring more states in an app.

As mentioned before, there are tools that reconstruct GUI models resulting from ripping and simulated user interactions. These approaches try to generate sequences of events under a particular strategy that drives and explores apps. [18]. Most of these tools generate finite-state machines with two purposes: (i) there is a need to generate sources of information to understand a system, whose models are nonexistent or too precarious. [25]; and (ii) model-based testing requires models to automate model generation processes.

Other approximations such as CrashScope [36] systematically explores Android apps and creates detailed crash reports in natural language. CrashScope enables context-aware input data and sensors and connectivity analysis. CrashScope makes contextual changes in the application based on API calls found statically in the code.

Contextual fuzzing is also an approach implemented in *Caippa* [29], a service for testing Windows mobile apps in a cloud environment. Contextual fuzzing refers to exercising apps with contexts observed in the wild (*e.g.,* eventual connectivity). In this case, GUI ripping is performed along the contextual fuzzing.

Injection of adverse conditions have been also explored not only with GUI ripping, but also using existing test cases. This is the case of *Thor* [1], a tool that injects unexpected events (*i.e.,* device rotation or incoming calls) in existing test suites.

Above all, previous research have used GUI models [32], [18]; combined usage and GUI models [34], [2], [27]; GUI and context models (partially) [36], [29]; and context information with existing tests [1]. However, none of the aforementioned approached have covered the multi-model vision we are proposing.

2.2 Hybrid applications

Hybrid apps are those in which developers write significant portions of their application in cross-platform web technologies, while maintaining direct access to native APIs when required [8]. This way of developing applications is gaining popularity among the mobile developers community because:

- Reduces the time to develop cross-platform apps
- It is based on Web technologies, which allows developers to recycle modules and components from existing Web developments
- Browser engines are becoming faster and mobile devices more powerful
- Performance differences between native and web technologies are becoming imperceptible

The industry leaders in hybrid applications are *React Native* and *Apache Cordova*, two different approximations with fundamental differences.

2.2.1 React Native

React native is a Facebook project that enables the construction of mobile apps based on React: a Javascript library. In this case, the code written in JS runs in a thread and do not cross-compile to other languages. To manage the GUI, React native uses the Android GUI framework, which means that graphical elements from an application built using this technology should be indistinguishable from the graphical components of a native application.

Given the fact that crawling and ripping Android applications is done through examination of the native graphical components, this kind of apps could be ripped used the existing GUI ripping tools aforementioned.

2.2.2 Apache Cordova

Apache Cordova is another technology that enables the access to native APIs from JS code. This piece of software gives access to Local Storage, Camera, GPS, and all the available sensors of each platform. Apache Cordova is the core of many popular hybrid frameworks such as Ionic, Adobe Phonegap, Monaca or Visual Studio.

The main difference from Apache Cordova to React Native is related to the way the GUI components are presented to the user. Contrary to React Native, Apache Cordova does not use native components. This technology encapsulates a Web Application based on JS, HTML and CSS into a Web View. Fig 2.1.

Because of the differences between React Native and Apache Cordova, some argument that React Native is not hybrid. Given the fact that React Native apps can be

Fig. 2.1: Contrary to native applications, Apache Cordova based applications contain just one activity. This activity has a basic layout with a central element: a *Web view*

crawled as native applications, henceforth, when we refer to hybrid applications we will refer exclusively to apps that use a Web View to present the GUI.

2.2.3 Testing hybrid applications

Most of the research conducted in hybrid applications is focused in two categories: Debugging and Security.

Debugging

The most popular tool for debugging hybrid applications and remote web views is Chrome Developer Tools [16]. These tools are directly built into the Google Chrome browser. Accessing to `chrome://inspect` displays the lists of web views in the devices connected through ADB. Once the browser is remotely inspecting the web view, all the errors, crashes and logs from JavaScript are printed in the computer's console. With Google Chrome, developers have access in real time to the DOM elements, network settings, and JS console to profile and audit the applications performance.

The first remote debugger was introduced in 2010: winre (WEb INspector REmote)[37]. It was part of the Apache Cordova project, however it is being deprecated due to the increasing functionality of the Google Developer Tools. There are tools based on Google Developer Tools that enhance debugging functionality, such as the Monaca Debugger[35] which include Cordova Plugins Supoprt and collaboration tools.

There is an approach to debug hybrid applications based on static analysis: HybriDroid[26]. It includes a built in bug detector based on a call graph which founds 4 types of bugs: MethodNotFound (when a JavaScript method call cannot find any target Java method to call), TypeOverloadedBridgeMethod (when JavaScript tries to use Java overloading), NotCompatibleTypeConversion (when a JavaScript type is not compatible with Java), and MethodNotExecuted (when a Java method returns an Array)[26].

Security

Tools and techniques related to security, are relevant to our approach because they examine the web view, and are able to analyze properties of hybrid applications.

This has been an important research topic in hybrid applications because of the possibilities of *cross-language code injection*. Jin *et al.* [19] define that this problem is inherited from Cross-Site scripting (XSS); additionally to their study, they developed a vulnerability detection tool. These vulnerabilities have been detected, and numerous tools to report them have been created: NOFRAK [9], Draco [43], [20]. Some of these approaches are based on instrumentation to reduce the attack surface [40]. BridgeTaint[4] is 'a bi-directional dynamic taint tracking method that can detect bridge security issues in hybrid apps' [4]. It analyzes privacy leaks and code injection when the app uses bridge communication dynamically. Another tool that addresses this problem is HybriDroid[26] which additionally to the bug detector, detects taints and leaks between Java and JavaScript. In comparison, BridgeTaint is able to detect security issues of apps built with frameworks, when HybriDroid sometimes do not.

Another topic related to security is the detection of SSL errors, Zuo *et al.* [47] present an approach to detect statically and dynamically vulnerabilities of this kind.

GUI ripping tools

As described above, GUI ripping tools for native apps are diverse and numerous, however, as of today, there is not a single approach that focuses on GUI ripping or crawling based on hybrid applications (based on a web view). Some of the tools mentioned in section 2.1.2 could work because of their implementation and the dynamic detection of components, nevertheless, they have flaws detecting different states and crashes.

3 Proposed Approach

Before describing the proposed approach we clarify the meaning of context, domain, usage, and GUI models. A domain model describes data, entities and their relationships in an application; it looks like a network of interconnected objects, where each object represents meaningful entities and concepts. Information in a domain model is useful to determine inputs and outputs in an application under test. A graphical user interface model (GUI) is also an informative model that represents all the graphic components and views of an application, and the events that trigger transitions among the views; it is commonly represented as a state diagram and it has been widely used for GUI ripping [3]. A context model [36] represents the surrounding conditions in which an app runs; in the case of mobile applications, it also includes sensors, networking, available hardware information (*e.g.,* device model, processor version), screen resolution and O.S version. An usage model describes how users can interact with an application and what functionalities are offered to them [32].

By augmented model (or multi-model) we mean the combination of the aforementioned models, in a single model that synthesizes relevant information. More formally, a multi-model is a directed graph. $G = (V, A)$, with V a set of states, in which each state has a unique combination of contextual variables, GUI elements and domain entities; and A a set of transitions, where each transition is a contextual change in the APP or an user interaction in the GUI that triggers a change in the app to a new state.

We illustrate the multi-model concept with the example presented in (Fig. 3.1, Page 14); the Figure depicts an abstraction for a multi-model generated dynamically and statically from a test app developed by the authors. The multi-model has 3 states (*S1*, *S2* and *S3*), and 2 edges (*T1* and *T2*). In this example, the app is impacted by contextual changes.

S1 is the home window of the application. It is a state that mixes graphical, usage and context information. *S1* occurs when Wi-Fi and cellular networks are active. On average, when this state is active, the device has an availability of 80% of its memory. From *S1*, the app flow can go to two states (*S2* and *S3*). The transition from *S1* to *S3* (*T2*) occurs when Wi-Fi is turned off. *S3* is a state running the same

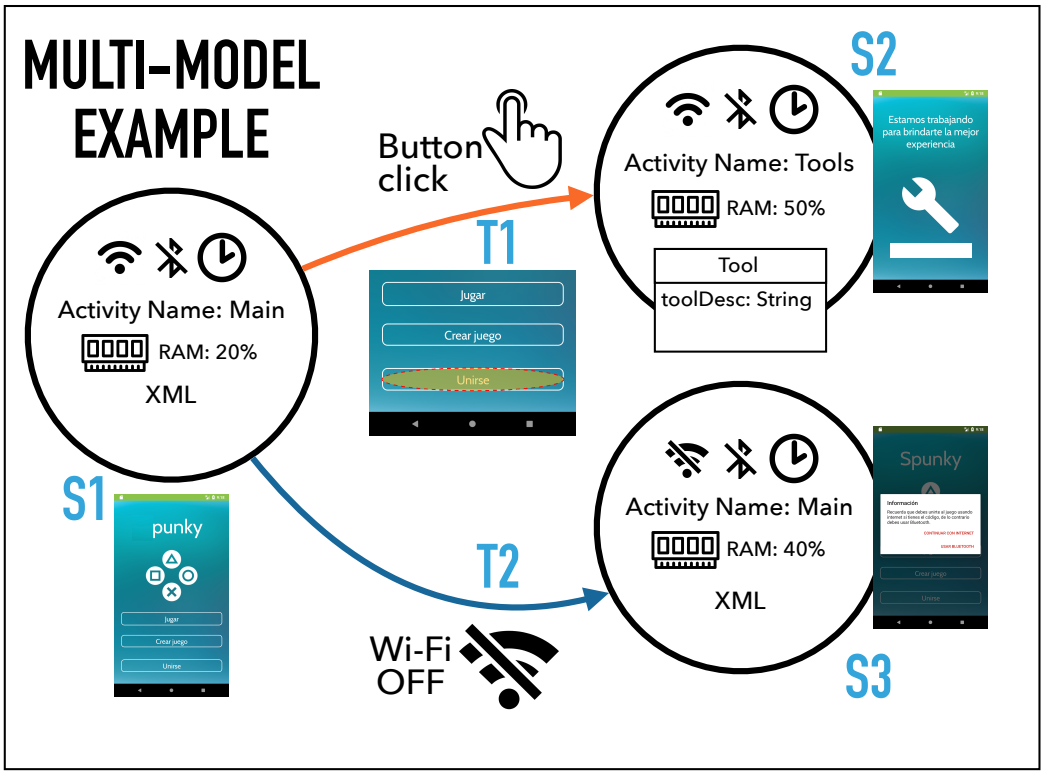

Fig. 3.1: Example of multi-model generated from an Android app. The states and transitions are a subset of the complete multi-model for the analyzed app.

activity as *S1*, however, it displays an alert message that fades out other buttons and the background. *S3* is more memory greedy than the initial state.

Transition *T1* changes app state from *S1* to *S2*. This transition is activated by clicking the button located at the bottom of the screen. Once this button is pressed, *S2* is activated. Context, graphical and usage information is also available in *S2*, however, it also includes domain-related information because there is a text-input for collecting information from the user; thus, we consider the activity (*i.e.,* the Android window related to *S2*) as an entity named "Tool" with a string field called "toolDesc".

3.1 General Approach

We have designed an architecture (Fig. 3.2, Page 15) that enables a series of stages that include: ripping, automatic extraction of static and dynamic models, generation of multi-models and model-based tests generation.

The multi-model generation starts by ripping the app under test. For this purpose, we have designed and developed a desktop tool called **RIP**, which is publicly available at

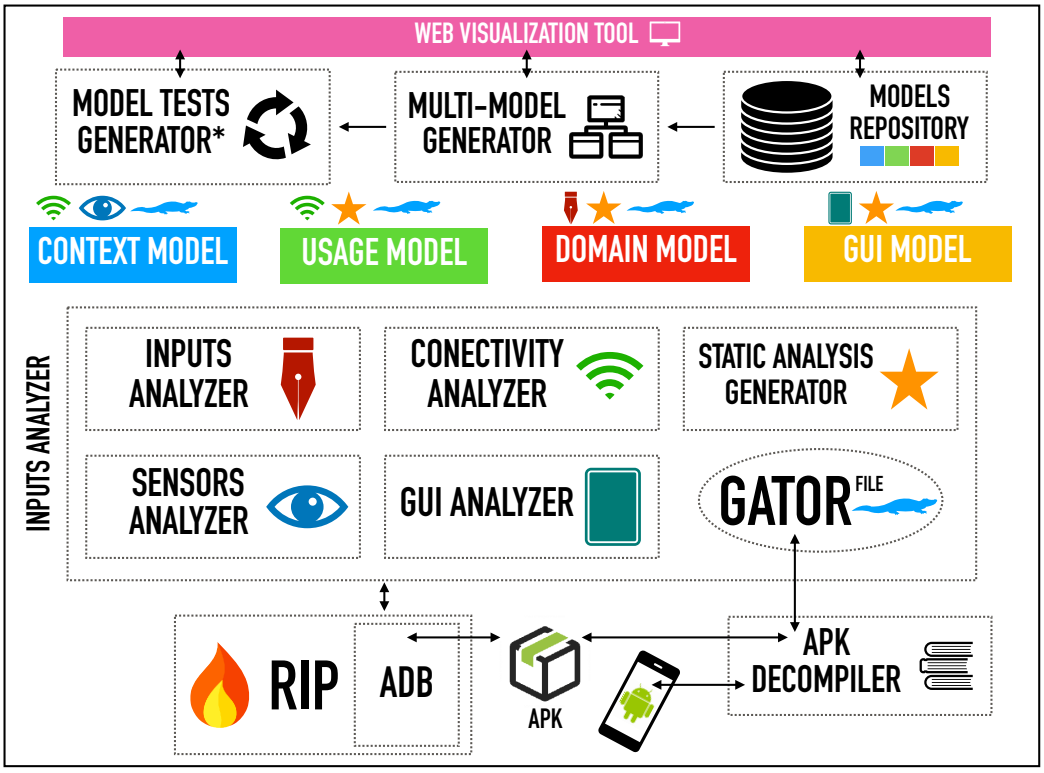

Fig. 3.2: Proposed architecture (**RIP**) for extracting augmented models for Android apps.

`https://github.com/TheSoftwareDesignLab/rip`. It is an automation software that executes a series of actions and emulates user interactions into an Android device to extract models; this is done through the Android Debug Bridge (ADB) [10], a command-line tool that provides access to an Android device over USB or Wi-Fi. In comparison to monkey/random testing tools that generate random interactions to find bugs and corner cases, **RIP** follows an strategy to explore the application based on the dynamic content that appears on a device screen (like other rippers do). **RIP** not only simulates user interactions in the screen; **RIP** also executes contextual changes to the application that vary the network configuration and the readings of the sensors (*e.g.,* accelerometer, gravity, gyroscope, light, proximity, magnetic field). During the execution, **RIP** collects GUI-related, domain-related. sensors-related, and resources-related information. In addition, screenshots and GUI-hierarchies are collected for each state.

Our approach differs from existing ones because we are able to (i) generate a comprehensive list of contextual changes, (ii) extract a domain model from GUI states, (iii) augment the dynamically-generated model with information collected statically, and (iv) considers exploration of hybrid applications. Due to security restrictions imposed by Android devices and the Android framework, in order to enable contextual events execution via ADB (*e.g.,* airplane mode, Wi-Fi), it is necessary to run the commands in a rooted physical device or an emulator.

RIP takes advantage of static code analysis to enrich the final generated model. When the ripping process finishes, RIP augments the collected model with a graph generated statically by GATOR [44], a static reference analyzer tool for GUI objects in Android. This tool has been chosen because its context-aware approach in static code analysis [45]. GATOR finds static references to GUI components and determines its control flow. If **RIP** identifies missing states (*i.e.,* states detected by GATOR but not by **RIP**), it adds the new states to the dynamically generated model. Thus, the GUI and usage models are represented by the combination of transitions and states information extracted from the ripping and GATOR.

GATOR code is not included in **RIP**. To combine static information with RIP's execution, GATOR analysis should be run first, and then passed to **RIP**. An example of the file that **RIP** imports from GATOR is presented in the code example 3.1

Code example 3.1: Fragment of a GATOR file obtained from a native application

```
{"nodes": [{
"id": 2160,
"name": "com.simplemobiletools.commons.activities.FAQActivity"
},{
"id": 2076,
"name": "android.view.Menu"
},{
"id": 2140,
"name": "android.view.Menu"
},{
"id": 14106,
"name": "android.app.DatePickerDialog"
}],
"edges": [{
"source": 2076,
"target": 2140,
"event": "implicit_power_event"
},{
"source": 14106,
"target": 14106,
"event": "implicit_home_event"
},{
"source": 2140,
"target": 2140,
"event": "implicit_rotate_event"
},{
"source": 2160,
"target": 2160,
"event": "implicit_home_event"
}]}
```

3.2 RIP components

Our architecture defines a layer of analyzers that guide **RIP** during the models extraction:

3.2.1 GUI analyzer

It extracts the hierarchy of graphical components in the app. It is able to differentiate app GUI states based on the analysis of the GUI hierarchy represented as an XML file. It means, the construction of the GUI model is done iteratively, according to the app exploration. To determine if two views are different, a decision process is followed. Firstly, if the activity names differ from each other, the two views are classified as different states. Otherwise, if the activity names are the same, then the XML is analyzed; each view is compared by the number of elements, checked boxes, buttons, labels content, alerts, and messages. In the case of menus or dialogs that are displayed on a view, we consider them also as states.

Code example 3.2: XML dump from an app, containing the layout hierarchy

```
1 <?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
2 <hierarchy rotation="0">
3 <node index="0" text="" resource-id="" class="android.widget.FrameLayout" package="com.
      clockwork.mcdonalds"
4 content-desc="" checkable="false" checked="false" clickable="false" enabled="true"
      focusable="false" focused="false"
5 scrollable="false" long-clickable="false" password="false" selected="false" bounds="
      [0,0][1080,1794]">
6 <node index="0" text="" resource-id="" class="android.widget.LinearLayout" package="com.
      clockwork.mcdonalds"
7 content-desc="" checkable="false" checked="false" clickable="false" enabled="true"
      focusable="false"
8 focused="false" scrollable="false" long-clickable="false" password="false" selected="
      false" bounds="[0,0][1080,1794]">
9 <node index="0" text="" resource-id="android:id/content" class="android.widget.
      FrameLayout" package="com.clockwork.mcdonalds"
10 content-desc="" checkable="false" checked="false" clickable="false" enabled="true"
      focusable="false"
11 focused="false" scrollable="false" long-clickable="false" password="false" selected="
      false" bounds="[0,63][1080,1794]">
12 <node index="0" text="" resource-id="" class="android.widget.ImageView" package="com.
      clockwork.mcdonalds"
13 content-desc="" checkable="false" checked="false" clickable="false" enabled="true"
      focusable="false"
14 focused="false" scrollable="false" long-clickable="false" password="false" selected="
      false" bounds="[0,63][1080,1794]" />
15 </node>
16 </node>
17 <node index="1" text="" resource-id="android:id/statusBarBackground" class="android.view.
      View" package="com.clockwork.mcdonalds"
18 content-desc="" checkable="false" checked="false" clickable="false" enabled="true"
      focusable="false"
```

```
19 focused="false" scrollable="false" long-clickable="false" password="false" selected="
       false" bounds="[0,0][1080,63]" />
20 </node>
21 </hierarchy>
```

The set of states gathered from visual information are the source for the GUI model. **RIP** captures image screen-shots and XML layouts from every state it has found. It also records the interaction/event that triggered the state transition (*e.g.,* pressing a button); note that state transitions are modeled as edges connecting states. To extract the XML layout information, **RIP** executes the command `adb shell uiautomator dump`. Eventually, the device stores this dump into an XML file such as the code example 3.2. Because this file is stored in the SD card, it must be extracted and parsed in the host computer. To capture the image snapshot of the device, a remote screen-shot is invoked with the command `adb shell screencap`. All the files in the devices are extracted through `adb pull`.

3.2.2 Inputs analyzer

It enables **RIP** to identify user input-related GUI Android components and interact with them accordingly (*e.g.,* check boxes, text boxes, lists, scrollable views). **RIP** creates random input data for the components based on the input types defined for text fields, and the component nature (*e.g.,* a check box can be activated or deactivated). Keyboards that appear on the screen and GUI components meta-data, give us clues about concepts and domain entities that are part of the domain model of the application. We define entities of the domain model as views that have input elements. Each entity has a set of attributes that correspond to input-related components in the view, and the attributes type is inferred from the type of input allowed (*e.g.,* numbers, special characters, boolean check, lists, *etc.*).

3.2.3 Sensors analyzer

During the ripping, **RIP** turns off/on sensors, randomly. The sensors analyzer identifies which sensors the application has access to from the app's manifest file. Once the sensors analyzer has this information, it detects the states where sensors are on and off using specific ADB commands.

To obtain a detailed list of sensors, **RIP** invokes the command `adb shell pm list features`. The available sensors in the device are listed as features (*e.g.,* `hardware.camera`, `hardware.sensor.gyroscope`, `hardware.sensor.light`)

3.2.4 Connectivity analyzer

This component identifies connectivity status of the app during all the ripping execution. Once the GUI has detected an initial set of states, it sends requests to the device in order to control Bluetooth, Wi-Fi, celullar networks and airplane mode. **RIP** triggers connectivity changes in every view of the application. If an error occurs or the GUI changes after the contextual change, then a new state is discovered and added to the model.

To determine the connectivity settings of the device, **RIP** calls the command `adb shell dumpsys wifi | grep 'Wi-Fi is'` to determine Wi-Fi status, `adb shell dumpsys wifi | grep 'mAirplaneModeOn'` to determine Airplane Mode, etc. The activation or deactivation of these settings is done with the commands `adb shell svc wifi (disable|enable)`, `adb shell svc bluetooth (disable|enable)`, `adb shell svc data (disable|enable)`, etc. Commands that enable or disable connectivity settings of the device require root privileges.

3.2.5 Static analyzer and GATOR

Different from the other components, it imports a flow graph of the app based strictly on static analysis, by relying on the GATOR tool [44]. GATOR creates a window transition graph with activities, dialogs and menus. The transitions of the graphs include events such as button pressings and window stack operations (push and pop). This flow graph is used to augment the model collected dynamically by **RIP**, in particular to have a more comprehensive list of transitions and states. This graph is presented in the code example 3.1.

3.2.6 RIP GUI

It interacts directly with ADB to explore the applications dynamically and coordinates the execution of the other components. The **RIP** GUI combines the ripping, interactive data collection, and static analysis, to generate individual models and a multi-model like the one presented in (Fig. 3.3, Page 20); the **RIP** GUI also generates the model as a JSON file that can be analyzed by any other tool.

3.3 Ripping hybrid applications

As described before, hybrid apps are those in which developers write significant portions of their application in cross-platform web technologies, while maintaining

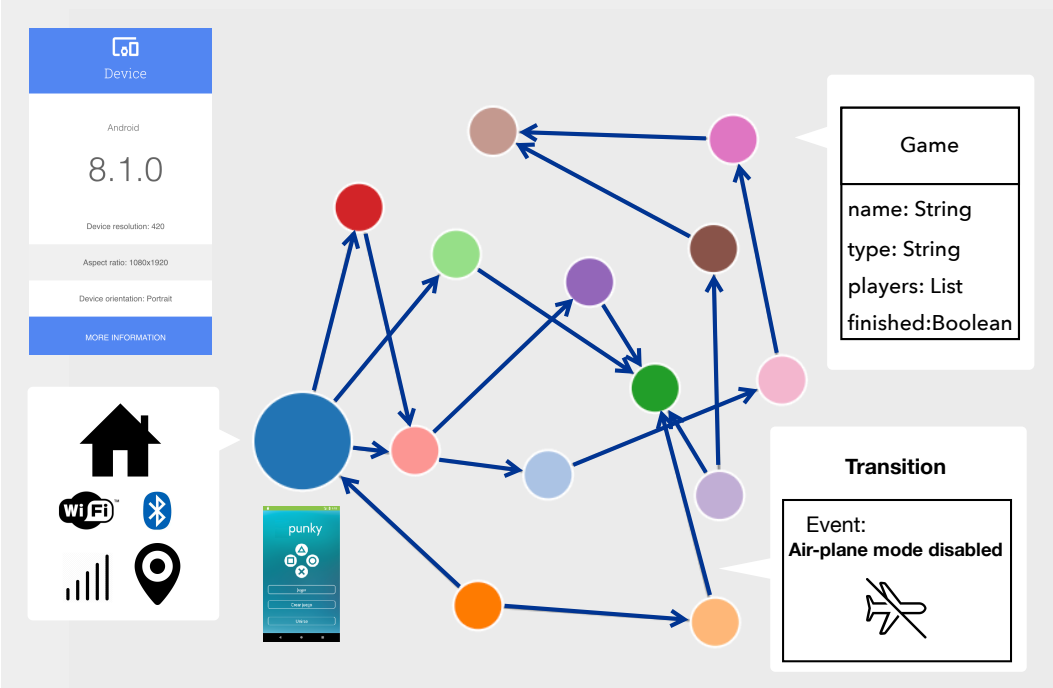

Fig. 3.3: Example of multi-model created by RIP.

direct access to native APIs when required [8]. This way of developing applications is gaining popularity among the mobile developers community because:

- Reduces the time to develop cross platform apps
- It is based on Web technologies, which allows developers to recycle modules and components from existing Web developments
- Browser engines are becoming faster and mobile devices more powerful
- Performance differences between native and web technologies are becoming imperceptible

Currently, Apache Cordova is the component that enables the access to native iOS and Android APIs from the JS code. This piece of software gives access to Local Storage, Camera, GPS, and all the available sensors of each platform. Apache Cordova is the core of many popular hybrid frameworks such as Ionic, Adobe Phonegap, Monaca or Visual Studio.

3.3.1 Barriers for ripping hybrid apps

Ripping native apps is different from ripping hybrid apps. The main reason is the underlying techonologies; in native apps the GUI is rendered by the view system and

window manager of the framework, in hybrid apps the GUI and events are managed by a Web View. In native applications, a native layout defines the structure of the user interface. An activity contains a hierarchy of containers and graphical objects called *Views* and *View groups*. These objects are also known as *widgets*. Widgets could be instanced as elements such as buttons, labels, text fields, among others. View groups provide the structure of widgets and other view groups in the screen [13]. To crawl dynamically apps, this hierarchy must be extracted continuously, and based on the layout information, the ripper can execute commands and simulate user interactions.

Contrary to native applications, most hybrid applications contain just one activity. This activity has a basic layout with a central element: a *Web view* [17]. Web views are widgets that can render web content (Fig. 2.1, Page 9). When the user interacts with elements of the web view, the JavaScript engine is in charge of changing the view. All the application logic is entirely written in JavaScript, therefore, errors, bugs and crashes are only visible to the web console. Additionally, the HTML DOM is not visible through the `adb dumpsys` command.

3.3.2 Strategies for ripping hybrid apps

In **RIP**, we have defined and implemented a series of strategies designed to tackle the ripping of hybrid apps. Specifically, we consider important to maximize the coverage of states discovered and enable detection of crashes, while maintaining the strategy of multi-model extraction and contextual exploration.

Detecting crashes in hybrid apps

Crashes are usually the result of uncaught exceptions. Other crashes come from ANRs (*Application Not Responding*), permission denials, network errors, and bad practices. These kind of crashes are well defined in the Android Developers Documentation and if one of them occurs, it is reported in the device. When one of these crashes is thrown, rippers of native apps are able to recognize them because the device writes them in the device log (logcat). In general, hybrid applications do not deal with these errors as native applications do, because their errors are written in the JS console. In order to capture errors in hybrid applications, RIP reads continuously the WebView console with the help of Chromium.

While **RIP** crawls a hybrid application, it listens to the web console, and stores the web errors associated with each state. As these errors occur in an embedded web browser, all the HTTP errors can be associated to the running application.

Maximize the coverage of states discovered

In order to maximize the coverage of states discovered, **RIP** must interact with every 'clickable' element inside the web view. The exploration should not be focused on finding and interacting with Android widgets, but with every new component accessible in the web view. A better approximation to explore the hybrid GUI should extract and parse the DOM of the web components with the help of the Chrome inspector, however, this is considered as future work.

We found experimentally that introducing some randomness to the ripping process in these applications improves the discovering of new states, because not all the web elements can be extracted solely with UIAutomator, and always remains actions of the application that could not be accessed systematically.

Finally, a key step that improves exploration of hybrid apps is triggering contextual changes, specifically, toggling network and Internet settings. This occurs because sometimes, all the web view content is not served just from the mobile device, but also from external servers in the Internet. Hybrid applications are highly prone to errors due to contextual changes.

Code example 3.3: Pseudocode describing the Ripping process

```
transitions = []
discoveredStates = []
contextualChanges = [rotateScreen, turnOnWifi, ...]
func explore(t):
  s.clickOrExecute()
  transitions.push(s)
  if thisIsANewState():
    discoveredStates.push(currentState)
  if state.hasClickableElementsLeft():
    for everyClickableElementInState c:
        if c has not been clicked:
            explore(c)
  elif !state.hasClickableElementsLeft() and !state.contextualChangesDone():
    for every cc in contextualChanges:
      explore(cc)
  elif state.ContextualChangesDone()
    if state.isHybrid():
      enableJSConsole()
      action = randomAcctions()
      explore(action)
```

4 Empirical study

The *goal* of our empirical study is to evaluate the proposal of combining multiple models to improve accuracy of testing processes and to evaluate **RIP** as a tool to automatically crawl Android applications. We evaluate our proposal in terms of (i) the importance of combining orthogonal information from different models and (ii) the suitability of a multi-model testing tool, in the process of developing mobile applications. At the same time, we evaluate **RIP** in terms of (iii) RIP's ability to crawl native and hybrid Android apps, and (iv) RIP's ability to detect bugs and generate crashes. The *context* of this study consists of (i) a set of 15 open source native APKs, (ii) a set of 20 hybrid APKs from the Google Play Store, and (iii) three baseline approaches for extracting models and crawl Android applications: DroidBot [27], Firebase Test Lab Robo [12] and Monkey [15].

The *quality focus* of this study is the effectiveness of **RIP** to detect states on both native and hybrid apps, reporting crashes and generating multi-models from these applications. To aid in achieving the goals of the study, the following research questions were formulated:

- ***RQ***$_1$: *Is the combination of multiple models useful to gather more information of an app under test?*
- ***RQ***$_2$: *How accurate is the state discovery algorithm' implemented in RIP when compared to state-of-the art tools?*
- ***RQ***$_3$: *Is RIP suitable to detect crashes and bugs in Android apps?*

Note that when answering the research questions we take into account hybrid and native Android apps. These two development strategies could be indistinguishable for final users of the apps, yet their are very different to crawl, analyze and build. **RIP has been designed to support both types of apps.**

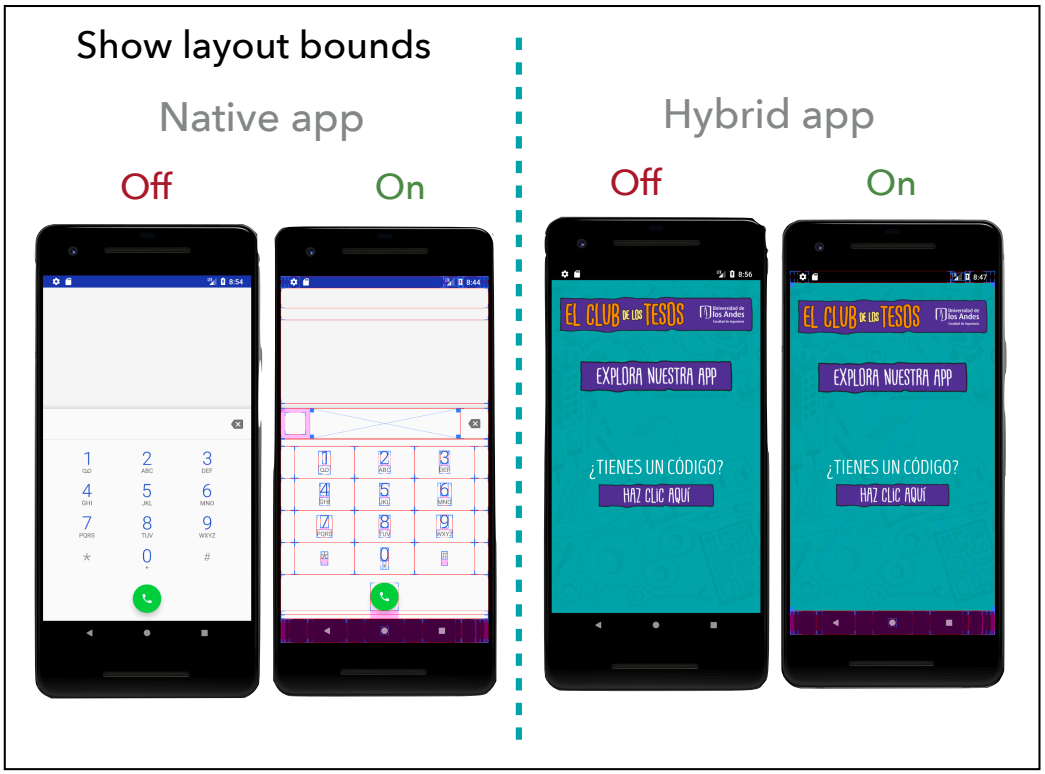

Fig. 4.1: Enabling layout bounds detection in hybrid and native apps

4.1 Context of the study

A set of 15 native Android apps and 20 hybrid apps were downloaded from the Google Play Store and F-Droid. The native apps were randomly selected from F-Droid. In order to identify and download the set of hybrid apps, it was necessary to check the different hybrid frameworks showcases, find the APKs in the Google Play Store and then decompile the APKs to identify the presence of hybrid packages (*e.g.,* `com.ionicframework`). Additionally, to verify that these apps contain web based GUI elements, the *Show layout bounds* setting was activated in a device. With this option enabled, it is possible to visually identify native Android components in a device screen. Figure 4.1 shows that in native apps, this option draws red and blue lines around every graphical element. On the other hand, it can be seen that when this option is enabled in a hybrid app, none of these components is detected by the layout manager.

All the applications were installed and tested in an Android emulator for a *Pixel 2* device with API 27 (Android 8.1 Oreo). The screen proportions of the device are `1080 x 1920` with `420dpi`. The CPU/ABI of the device is a Google's API Intel Atom (x86) with 1.5 GB of RAM.

Tab. 4.1: List of applications used in the study

Name	Category	Size(MB)	Source	Tech	Package
Amaze	Tools	5.8	F-Droid	Native	com.amaze.filemanager
aMetro	Maps & navigation	2.62	F-Droid	Native	org.ametro
Barcode Scanner	Tools	0.73	F-Droid	Native	com.google.zxing.client.android
Materialistic	News & Magazines	3.56	F-Droid	Native	io.github.hidroh.materialistic
Minimal	Productivity	1.88	F-Droid	Native	com.rubenroy.minimaltodo
Mysplash	Personalization	4.82	F-Droid	Native	com.wangdaye.mysplash
Omni Notes	Productivity	4.82	F-Droid	Native	it.feio.android.omninotes.foss
RadioDroid	Music & audio	4.19	F-Droid	Native	net.programmierecke.radiodroid2
Tasks	Productivity	6.29	F-Droid	Native	org.tasks
Car Report	Auto & Vehicles	2.93	F-Droid	Native	me.kuehle.carreport
Calendar	Productivity	6.08	F-Droid	Native	com.simplemobiletools.calendar
Tusky	Social	2.93	F-Droid	Native	com.keylesspalace.tusky
AnotherMonitor	Tools	0.2	F-Droid	Native	org.anothermonitor
Antennapod	Video Players	5.87	F-Droid	Native	de.danoeh.antennapod
Trolly	Shopping	0.03	F-Droid	Native	caldwell.ben.trolly
McLaren Automotive	Auto & Vehicles	60	Google P.	Hybrid	com.mclaren.mclaren2019
Joule	Food & Drink	95.3	Google P.	Hybrid	com.chefsteps.circulator
Sworkit	Health & Fitness	78.1	Google P.	Hybrid	sworkitapp.sworkit.com
Untappd	Food & Drink	44.4	Google P.	Hybrid	com.untappdllc.app
Hockey Community	Sports	4.9	Google P.	Hybrid	com.hockeycommunity.hc_app
Tomatoid	Productivity	0.3	Google P.	Hybrid	com.tomatoid.api
Mobimall	Libraries & Demo	16.7	Google P.	Hybrid	com.verbosetech.mobimall_ionicdemo
Ionic Framework	Education	4.7	Google P.	Hybrid	com.gsolution.ionicdemo
El club de los tesos	Education	15.8	Google P.	Hybrid	uniandes.tsdl.tesosApp
IELTS PRACTICE	Education	2	Google P.	Hybrid	com.examgroupapps.ieltslistening_practice
TripCase	Travel & Local	12.5	Google P.	Hybrid	com.sabre.tripcase.android
Pacifica	Tools	38.2	Google P.	Hybrid	icom.pacificalabs.pacifica
Tripline	Travel & Local	1.9	Google P.	Hybrid	inet.tripline
iTasca	Food & Drink	3.8	Google P.	Hybrid	it.tascadalmerita.iTasca
EcoAlimentate	Health & Fitness	1.5	Google P.	Hybrid	com.rma.ecoalimentate
Folver	Education	1.8	Google P.	Hybrid	com.megabyteraingmail.com.theformulasolver
Hybrid Native	Libraries & Demo	4	Google P.	Hybrid	com.kathanshah.hybridnative
McDonald's	Food & Drink	9.6	Google P.	Hybrid	icom.clockwork.mcdonalds
Cordova ionic VR	Shopping	23.1	Google P.	Hybrid	it.tangodev.cordovapluginvrviewsampleapp
Fashion Ecommerce	Libraries & Demo	11	Google P.	Hybrid	icom.vectorcoder.ionicecommerce.demo2

The detailed list of applications used during the study is presented in Table 4.1

4.2 RQ_1 Combining multiple models to improve accuracy of testing processes

To answer RQ_1, we conducted a case study to show that combining the different models in an augmented model could improve the accuracy of testing processes. It means, we wanted to understand whether combining the models is useful to gather more knowledge of an app under test, and whether that knowledge could be used to generate more robust test cases. To this, we (i) extracted multi-models from 5 different Android apps which are listed in (Table. 4.2), and (ii) collected the following information using RIP:

- Execution time required to explore the applications until no more states were discovered,

Tab. 4.2: General results obtained from multi-model extraction. **Abbreviations for column headings**. CR = Car Report, YLC = Your Local Weather, SC = Simple Calendar

	CR [24]	Punky	YLC [41]	Tasks [5]	SC [42]
Execution time (mins.)	4.6	4	6	5.3	4.8
Total number of states discovered	51	30	31	37	36
States discovered due to contextual changes	0	8	5	4	4
Domain entities extracted	21	4	23	17	20
Attributes extracted	63	96	27	60	44

- Total number of states discovered by triggering simulated user interactions and context events (ripping),
- States that were discovered due to contextual changes (ripping + context),
- Domain entities extracted from the apps,
- Attributes extracted from each domain entity

In the study we executed **RIP** in two modes: (i) ripping only mode, and (ii) ripping + contextual model execution. As reported in (Table. 4.2), when we added the contextual model, we found new states in 4 out of 5 applications tested. This suggest that generating augmented models can improve software comprehension and testing tasks for mobile apps, because different states are activated by combining individual models.

The first row of (Table. 4.2) shows the execution time required to explore the applications until no more states were discovered. Second row includes total number of states discovered by triggering simulated user interactions and context events. Next row shows only the states that were discovered due to contextual changes. Based on the information of these two rows is easy to conclude that Car Report is an application that relays much less to contextual changes in comparison with Your local weather or Simple Calendar. In the case of Punky, changes due to context were triggered by Wi-Fi, Bluetooth and accelerometer interactions.

The Domain entities extracted' row refers to entities discovered. This number is directly correlated to the number of application views that contain text inputs, selectors and checkboxes. Each entity contains a set of attributes, whose sum correspond to the final row.

Table 4.2 shows that information from each model is complementary and orthogonal. For instance, applications that are highly dependent on sensors and networking connections have states that could not be discovered if contextual events are not considered.

***RQ*$_1$** Is the combination of multiple models useful to gather more information of an app under test?

Automatically extracting augmented models from Android apps enables better understanding of the apps. For modern mobile applications, ripping apps — but based only on GUI exploration— is not enough because they are context-aware. To that end, context, GUI, usage and domain models should be extracted and combined together to build more useful and comprehensive augmented models.

4.3 **RQ**$_2$ How accurate is the *state discovery algorithm* implemented in RIP when compared to state-of-the art tools?

A second case study was conducted to determine how RIP compares with existing tools in industry and academy. Table 4.3 depicts the features of the tools. The state discovery algorithm of RIP includes (i) contextual changes, (ii) DFS based exploration and detection of hybrid or native components in applications, (iii) triggering monkey actions in the user interface and (iv) detection of states. Code example 3.3 describes this algorithm.

Tab. 4.3: Industry and academy tool features

	RIP	**Monkey**	**Firebase**	**DroidBot**
GUI Ripping	✓		✓	✓
Contextual changes	✓			
Random events	✓	✓	✓	✓
Model-based crawling (systematic)	✓		✓	✓
Works without instrumentation	✓	✓	✓	✓
State graph generation	✓		✓	✓
Crashes detection	✓		✓	

4.3.1 UI/Application Exerciser Monkey

'Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events' [15]. Monkey is part of the Android SDK and run from the command-line. Monkey was picked for the study because it (i) testers use it frequently to stress applications, (ii) it is part of the Android SDK and (iii) reports crashes and errors. 'Is the most frequently used tool to test Android apps, partly because it is part of the Android developers toolkit and does not require any additional installation effort' [7]

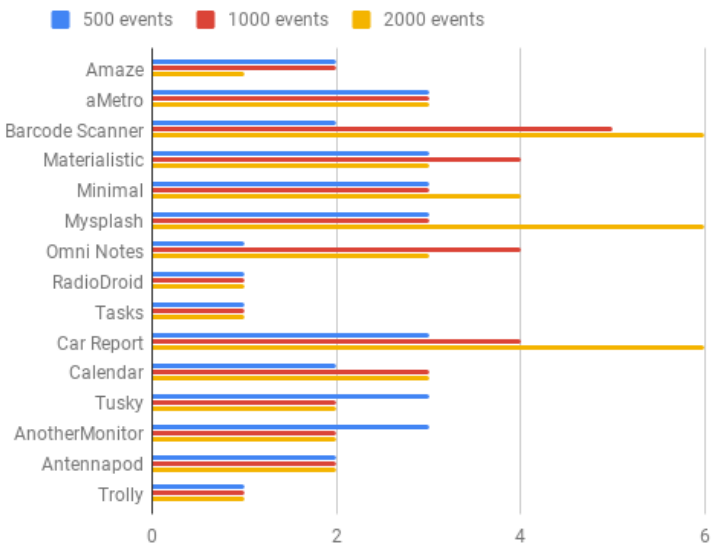

Fig. 4.2: Number of discovered activities by Monkey executions with 500, 1,000 and 2,000 events over native applications

The UI/Application Exerciser Monkey was executed with native and hybrid applications. Using the command `adb shell monkey -p <Package name> -throttle <Time between events> -v <Number of events>`. For each application, 500, 1000 and 2000 events were executed in order to identify a relation between this number and the number of states discovered.

Native applications. The results of Monkey executions over native applications are presented in Figure 4.2. The average of discovered activities with 500 events was 2.2, with 1,000 events was 2.6 and with 2,000 events was 2.9. In order to determine if there is a difference in the number of activities found for each quantity of events, as data was obtained from the same applications and sample sizes are small (n=15), it must be performed a nonparametric test for related samples therefore, a Friedman test was performed to compare the three series.

The null hypothesis for the test is that there are no difference between the number of activities found by Monkey with 500, 1,000 and 2,000. The alternative hypothesis is that at least with one of the number of events changes the number of activities found by Monkey.

The p value obtained was 0.42 hence the null hypothesis is not rejected, using an alpha of 5% it was concluded that there is no difference in the number of activities explored by Monkey ranging by 500, 1,000 and 2,000 events.

Hybrid applications. Even though Monkey does not advertise special capabilities for hybrid applications, it was able to execute random events in the user interface.

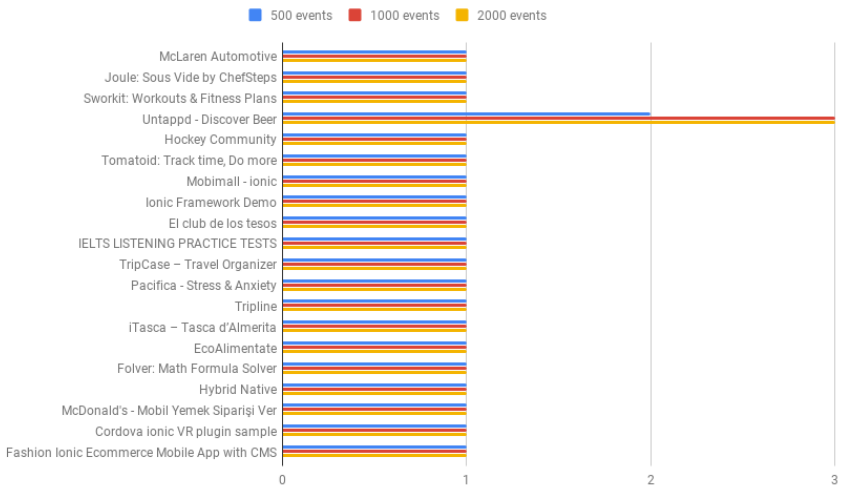

Fig. 4.3: Number of discovered activities by Monkey executions with 500, 1,000 and 2,000 events over hybrid applications

During the tests execution, it was evident that these random interactions made changes in the interface. Monkey is not designed to perform contextual changes in the application, however, due to the randomness of the GUI changes, in some rare cases it turned off or on airplane mode and connectivity settings.

Figure 4.3 Presents the discovered activities by Monkey with 500, 1000 and 2000 events. What this graph shows, is that almost for every app, the number of discovered activities was the same: 1. The explanation of this behavior comes from the conception of the hybrid app, with a single activity with a web view. The applications that contains more than a single activity are special cases that have multiple web views in different activities.

This experiment shows that even though Monkey made changes in the application, it only considers new activities as new states, it does not have a states memory. Monkey was successful interacting with the app, however, for hybrid apps, there is no way to determine how many states it visited if is not complemented with additional software.

Summary for UI/Application Exerciser Monkey. The experiments show that Monkey is capable of identify different activities in native applications. However, changing the number of events performed does not make a significant difference in the number of activities encountered. Also, the average activity coverage of Monkey is 38% in native applications, which suggests that a completely random discovery approach could be improved by a model-based approach.

As for hybrid applications, Monkey is not capable of reporting the number of states discovered insomuch as it only report activities. Monkey lacks ripping capabilities and does not aggregate results, which make analysis of its execution a hard task; a developer must follow all the events and activities to determine the states discovered by this random tool.

Based on the experiments conducted with this tool, we determined the importance of trigger random events in hybrid applications. This is a qualitative conclusion and tests to analyze quantitatively the importance of randomness in hybrid applications is part of our future work.

4.3.2 Firebase Test Lab Robo Test

Firebase Test Lab [12] is one of the industry leaders in mobile testing. It has a cloud-based app-testing infrastructure that enables concurrent execution of tests with and without instrumentation. Firebase Test Lab Robo Test is one of their services: 'Robo test analyzes the structure of your app's UI and then explores it methodically, automatically simulating user activities' [12]. It cloud service allows testers to run Robo test on virtual and physical devices in parallel, detecting crashes and performance issues.

Firebase Test Lab was chosen in the study because it is the Google's flagship product in automated testing.

For each application, a Robo Test was configured and run. To configure each test, the developer must choose the specific device: in our case, *Google Pixel's 2 API 27* was selected. The test recorded the number of actions performed during the exploration, the number of activities covered, the number of distinct screens visited and the duration. Is worth remembering that Firebase Test Lab is a cloud service and all the infrastructure is managed by Google. To that end, this service is very easy to use and the only requirement for the final user is to have the APK that he wants to test.

Native applications. The native applications were submitted to the service, and after each execution finished, Google sent an email informing the final result for every test. (Table. 4.4) presents the results of running Firebase Test Lab, indicating the duration, number of actions, activities and screens for the applications. Figure 4.4 depicts the outcomes of the execution for a specific app, including a full video of the whole crawling process, aggregated statistics and a very detailed crawl graph. This graph shows the different screens connected with edges associated to clicks and button pressings. The applications that seem to have a low number of screens (*e.g.,* Trolly: 1, Omni Notes: 3) have login and authentication activities in the

Tab. 4.4: Results of running Firebase Test Lab on native applications

Application	Duration (s)	Actions	Activities	Screens
Amaze	314	268	1	120
aMetro	309	343	4	73
Barcode Scanner	258	204	7	27
Materialistic	301	87	6	28
Minimal	304	200	4	26
Mysplash	313	68	6	23
Omni Notes	25	8	1	3
RadioDroid	313	122	1	79
Tasks	303	227	6	90
Car Report	305	202	5	37
Calendar	304	205	6	64
Tusky	33	12	1	4
AnotherMonitor	318	133	4	19
Antennapod	314	141	5	59
Trolly	11	7	1	1

applications. In these scenarios, Testlab is unable to automatically create users or introduce credentials, and does not explore more than the initial screens.

Hybrid applications. Regarding hybrid applications, Firebase Test Lab's behavior was entirely different from its native execution (Table. 4.5). In almost all the applications, the number of activities discovered was 1 because hybrid apps usually do not have more than one activity. There are two groups of apps: those with a number of screens greater than 60 and those with a number of screens lower than 10. Examining carefully those applications with a big number of screens found, we identified that Firebase classified screens with animations as multiple screens; one of this cases is presented in Figure 4.5. In this figure, an animation is running in the background of the screen, the animation is continuously changing. The Robo Test is visually detecting updates in the screen, and as the frames change, it detects new screens of the application.

Summary of Firebase Test Lab Robo Test. Firebase Test Lab is a easy to use service that explored native and hybrid applications in a short time. It detected several different screen in hybrid applications, however, it did not explore correctly hybrid applications. In none of them it was able to detect the states correctly and build the crawl graph.

4.3.3 DroidBot

DroidBot [27] is a tool that uses a model-based strategy to automatically explore mobile GUIs. It generates inputs and a transition model between different states of

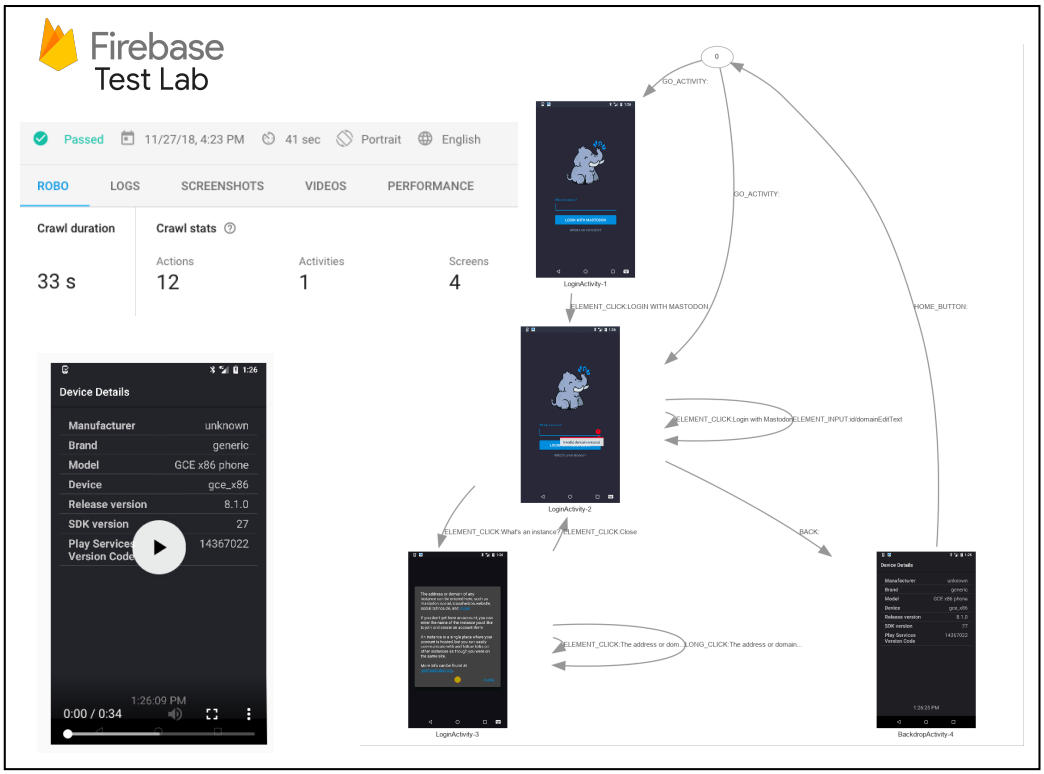

Fig. 4.4: Firebase Test Lab: Results of a Robo Test for a single native app

the application. One of the main features of DroidBot is that it does not require app instrumentation and is mean to run in almost any Android device. DroidBot is open source and publicly available [28]. It was choose as a baseline for the study because its availability and effectiveness exploring mobile applications.

DroidBot was executed running the command `droidbot <APK name> -o <Output folder> -is_emulator -random` for the complete set of applications. The `-random` flas was used to add randomness to input events and the `-is_emulator` flag was required for running droidbot on virtual devices . The exploration strategy used was *dfs_greedy* which explores UI using a greedy depth-first strategy. DroidBot implements other search strategies (*e.g., dfs_naive*, *bfs_naive*, *bfs_greedy*), but *dfs_greedy* was chosen because this strategy is a better approximation when trying to explore completely a tree.

For each tested application DroidBot generates a UTG (*UI Transition Graph*), screen shots of the states and an HTML report that includes the time spent, number of input events, number of UTG states, number of UTG edges and activity coverage.

Native applications. The results of DroidBot on native applications are shown in (Table. 4.6), the average activity coverage is 52% and the average duration is 73

Tab. 4.5: Results of running Firebase Test Lab on hybrid applications

Application	Duration (s)	Actions	Activities	Screens
McLaren Automotive	15	4	1	3
Joule	306	194	1	146
Sworkit	163	63	1	6
Untappd	9	6	1	2
Hockey Community	322	20	1	3
Tomatoid	86	70	1	3
Mobimall	9	8	1	2
Ionic Framework	37	10	1	3
El club de los tesos	11	4	1	3
IELTS PRACTICE	253	79	2	5
TripCase	26	15	1	3
Pacifica	282	235	4	62
Tripline	82	24	1	4
iTasca	79	17	1	3
EcoAlimentate	5	4	1	2
Folver	74	22	1	6
Hybrid Native	144	31	2	9
McDonald's	91	28	1	3
Cordova ionic VR	80	21	2	9
Fashion Ecommerce	72	17	1	2

minutes. Also, the number of input events rounds 990 and the discovered states range between 37 and 874 with a standard deviation of 218. It can be seen that the average duration of the test is more than 10 times the maximum duration of a Robo test in Firebase. Also, from the state graphs generated it can be noticed that many of the states registered by DroidBot are almost the same, they share the same layout but only the content in the fields changes. For example, the most repeated case is in a login form, each time DroidBot changes the input fields a state is reported. Moreover, if DroidBot leaves the login view and comes back after, it reports again all the states generated from login view.

Hybrid applications. The results of running DroidBot on hybrid applications are shown in (Table. 4.7), it can be noticed that even though there are multiple registered activities in most of the applications, only one is discovered by the tool. This is due to the fact that the hybrid applications use a Web View which encapsulates the GUI components using, in most of the cases, only one activity.

Moreover, the average duration of the test was 68 minutes, the number of input events ranges between 723 and 998, and the average number of discovered states is 94.1. However, from the state graphs it can be seen that as DroidBot is unable to analyze the Web View, it keeps exploring outer applications like Facebook or Google

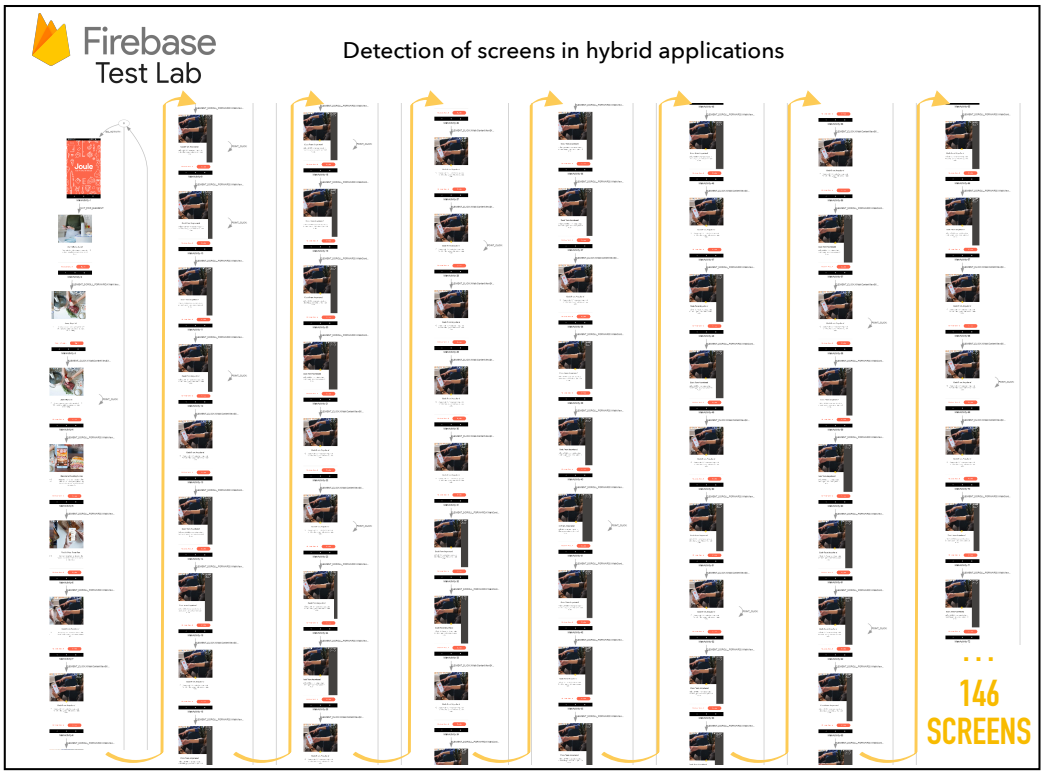

Fig. 4.5: For some hybrid apps with animations, Firebase detect the same state multiple times and reports a unreal number of screens

(accessing to these services in the authentication screen). Also, DroidBot does not report only different states, we manually counted the number of repeated and outer application states and reported them as false positives. It was estimated that the 67.1% of the states discovered correspond to false positives. As can be seen in Figure 4.6, extracted from a state graph generated by DroidBot, there are several states reported that share the same view.

Summary for DroidBot. DroidBot is able to generate a state graph from the ripping, it also reports the number of states discovered and its transitions. In native applications, it obtains an average activity coverage of 52% on native applications, which is higher to the obtained by Firebase (44%) and Monkey (38%). However, the time spent in the test is more than 10 times the maximum duration of a Robo test in Firebase, this opens the possibility to find new approaches of ripping that increase the activity coverage in less time.

As for hybrid applications, DroidBot is unable to analyze the Web View; therefore, it runs out of the main application generating a lot of states that does not refer to the application under test, and does not report the number of different states generating a large number of false positive states (67.1%).

Tab. 4.6: Results of running DroidBot on native applications **Abbreviations for column headings**. *RA* = Registered activities in the app, *FA* = Found activities during exploration, *AC* = Activity coverage, *Time(s)* = Time of execution, Input events = Number of input events generated, *UTG states* = UI Transition Graph States, *UTG edges* = UI Transition graph edges

App	RA	FA	AC	Time(s)	Input events	UTG states	UTG edges
Amaze	6	2	0.33	4492	996	295	453
aMetro	6	4	0.66	3966	998	108	248
Barcode Scanner	9	7	0.77	4173	997	49	106
Material Notes	23	6	0.26	5944	990	120	345
Minimal	5	4	0.8	3756	997	150	342
Mysplash	20	7	0.35	6796	995	221	489
Omni Notes	17	4	0.23	4238	992	196	405
RadioDroid	2	1	0.5	3959	990	453	548
Tasks	38	5	0.13	3437	997	69	116
Car Report	8	7	0.87	4607	997	254	402
Calendar	12	7	0.58	4068	996	227	313
Tusky	21	1	0.04	4754	996	16	37
AnotherMonitor	4	4	1	3437	995	819	874
Antennapod	20	7	0.35	4434	997	328	518
Trolly	2	2	1	3745	992	35	89

4.3.4 RIP

We also analyzed the apps with our proposed tool, **RIP**.

Native applications. In order to compare RIP with Firebase and DroidBot, Table 4.8 presents the average activity coverage, average number of states discovered, standard deviation of the number of states discovered, the average duration of the tests and the average number of new states discovered when applying contextual changes. It can be noticed that the higher activity coverage is reached by DroidBot (50.15%) however, its average duration is about 17 times higher than Firebase and 14 times higher than RIP. The average number of states found by DroidBot is 5 times higher than Firebase and 7 times higher than RIP. Nevertheless, neither DroidBot nor Firebase apply contextual changes in the exploration of states. With contextual changes RIP reported 6.33 new states discovered on average.

RIP exploration algorithm was behind Firebase and DroidBot, however, enabling contextual changes in exploration allows **RIP** to find states that these tools are not able to detect.

Monkey was not included in this comparison because Monkey only reports different activities reached and does not rip the applications, it only triggers random events during the execution, and records in a log the activities found and the events triggered. From monkey's execution, it was valuable to see that its random approach

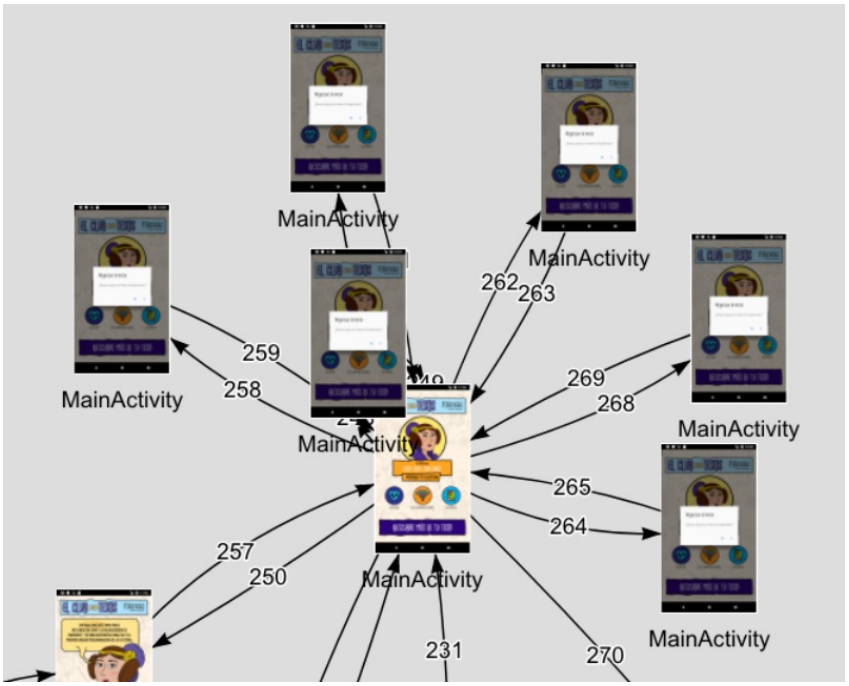

Fig. 4.6: Example of repeated states discovered by DroidBot

sometimes toggle rare settings, that were hardly discovered from systematic approaches. This random exploration is very useful to detect crashes, however, is not useful at all for building a multi-model because it is difficulty to reproduce large and unguided scenarios generated by Monkey.

Hybrid applications. In order to compare the state discovery algorithm implemented in RIP with Firebase and DroidBot (true positives only), we recorded the number of states discovered by each tool. As it can be seen in Figure 4.7, RIP detects a larger number of states almost in all applications compared to DroidBot and Firebase.

It can be observed from (Table. 4.9) that the average number of true positive states discovered by RIP is 50.1, without outer applications or repeated states. Also, the average duration of RIP was 7.31 minutes which is 10 times less than the average duration of DroidBot. The number of states discovered by Firebase is 3 times lower than RIP's. Finally, in hybrid applications the only tool able to do contextual changes and detect new states generated is RIP, and it found in average 5.6 new states with this changes.

The comparison of RIP vs Firebase performance is presented in Figure 4.8. RIP was able to extract a complete graph from the hybrid application explored, including all the views and screens of the application. Firebase Test Lab opened the application, detected the splash view, and then the main screen of the application. After that,

Tab. 4.7: Results of running DroidBot on hybrid applications **Abbreviations for column headings**. *RA* = Registered activities in the app, *FA* = Found activities during exploration, *AC* = Activity coverage, *Time(s)* = Time of execution, Input events = Number of input events generated, *UTG states* = UI Transition Graph States, *UTG edges* = UI Transition graph edges, *FP States* = False positive states discovered.

App	RA	FA	AC	Time(s)	Input events	UTG states	UTG edges	FP States
McLaren Automotive	6	1	0.16	3042	723	70	127	50
Joule	8	1	0.12	4792	997	65	120	49
Sworkit	14	2	0.14	8270	996	87	200	65
Untappd	11	2	0.18	3849	997	76	154	32
Hockey Community	17	2	0.11	3652	996	143	360	129
Tomatoid	1	1	1	3835	989	112	183	80
Mobimall	12	1	0.08	3508	997	174	388	155
Ionic Framework	1	1	1	3410	997	256	558	254
El club de los tesos	1	1	1	4035	997	32	99	14
IELTS PRACTICE	2	2	1	3782	998	86	229	79
TripCase	14	2	0.14	4072	992	142	240	122
Pacifica	10	2	0.2	4005	993	116	301	107
Tripline	3	1	0.33	3953	997	59	175	39
iTasca	1	1	1	4179	997	61	129	49
EcoAlimentate	1	1	1	3733	997	8	19	4
Folver	4	1	0.25	3654	997	32	93	15
Hybrid Native	1	1	1	3383	989	36	78	22
McDonald's	5	1	0.2	4012	997	28	63	10
Cordova ionic VR	3	1	0.33	3573	997	289	593	219
Fashion Ecommerce	18	1	0.05	5105	997	10	19	0

Tab. 4.8: Comparison in native applications between RIP, DroidBot and Firebase. **Abbreviations for row headings**. *Std states* = Standard deviation of the number of states discovered, *# States CC* = Number of states discovered when applying contextual changes.

Average metric	RIP	DroidBot	Firebase
Activity coverage	34.97%	50.15%	44.07%
# States	28.06	222.66	43.53
Std states	24.63	204.83	35.55
Duration (min)	4.93	73.11	4.13
# States CC	6.33	-	-

Firebase was unable to detect the components of the application. Related to hybrid apps exploration, RIP is superior.

***RQ*$_2$** How accurate is the *state discovery algorithm* implemented in RIP when compared to state-of-the art tools?

Still performance of **RIP** exploring native applications is lower than existing baseline tools' (based only on GUI ripping), enabling contextual changes in exploration allows **RIP** to find states that these tools are not able to detect. Our tool bridges the gap of ripping hybrid apps, and is the first approach that focuses in exploring dynamically these applications. **RIP** outperforms industry and academy state of the art tools in the exploration of hybrid apps in terms of detecting new states and not including repeated states in the crawl graph.

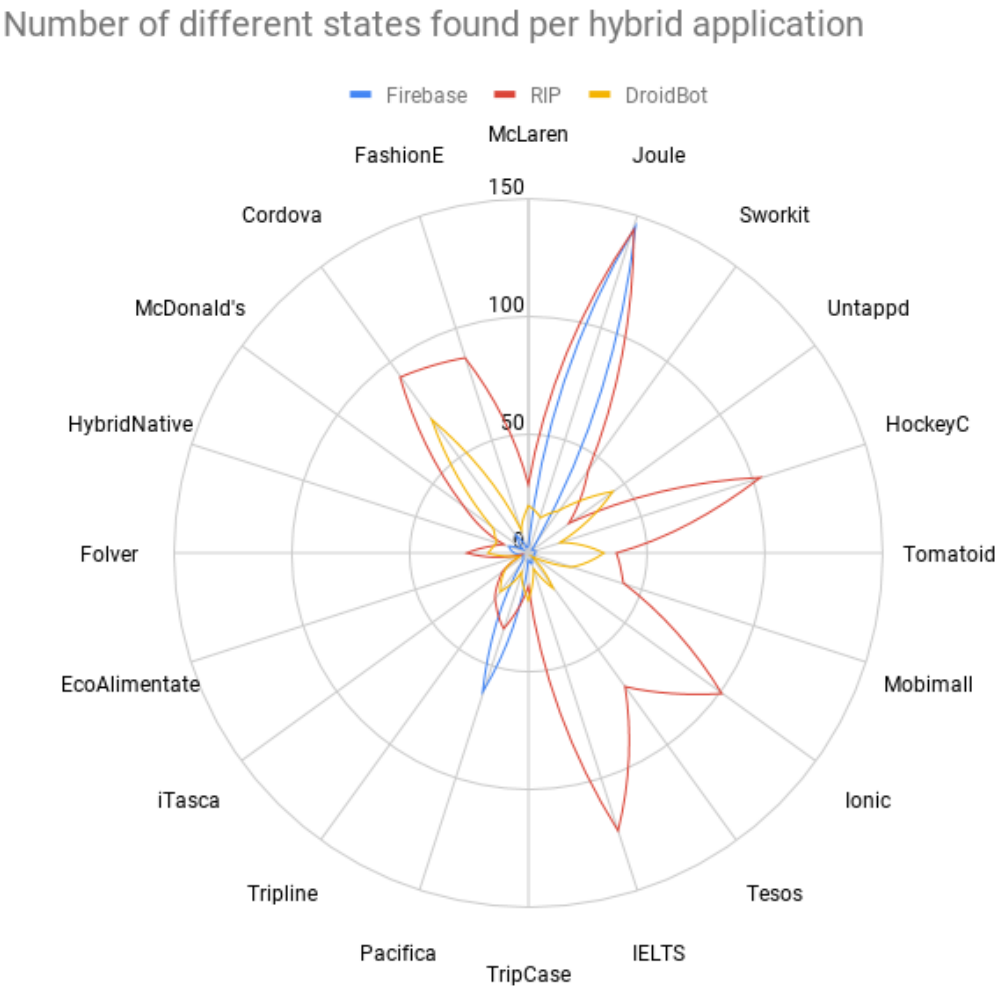

Fig. 4.7: Comparison of the number of different states discovered on hybrid applications between RIP, Firebase and DroidBot

4.4 RQ_3 Is RIP suitable to detect crashes and bugs in Android apps?

To answer RQ_3, a third case study was carried out in parallel to the experiments of RQ_2. While exploring each one of the applications, the number of crashes was obtained from **RIP**, Firebase and Monkey. Crashes from Monkey were extracted manually from the Monkey's log because it does not report aggregated results. In addition, DroidBot was not included in this case study because it neither report crashes nor bugs.

In the complete set of applications, Monkey found one crash in a native application. During the execution of Monkey in hybrid applications, it was able to triggered

Tab. 4.9: Comparison in hybrid applications between RIP, DroidBot and Firebase. **Abbreviations for row headings**. *Std states* = Standard deviation of the number of states discovered, *# States CC* = Number of states discovered when applying contextual changes, *# TP states* = Number of different states discovered, *FP states percentage* = Percentage of repeated or outer application or repeated states discovered.

Average metric	RIP	DroidBot	Firebase
# States	52.6	94.1	13.95
Std states	39.29	76.16	33.74
# TP States	50.1	19.4	12.7
Duration (min)	7.31	68.2	1.78
# States CC	5.6	-	-
FP states percentage	4%	67.1%	5%

events that caused ANRs and HTTP errors, however, it was not able to register them because Monkey only detect native crashes.

In native apps, RIP found 2 crashes related to contextual changes (turning on airplane mode) These crashes occurred when the device was connected to Internet and lost connection while executing a background download task. In hybrid applications, RIP found a series of crashes which are presented in the Figure 4.10. It found crashes in 40 % of the hybrid applications presented in the list. These crashes include HTTP 500 errors (Internal server errors without description), HTTP 503 errors (Service unavailable) and HTTP 404 errors (not found errors). Some of these crashes are depicted in Figure 4.9.

Firebase Test Lab was not able to find crashes in none of the hybrid applications, and did not discover the crashes in the native applications because they were triggered by contextual changes.

We realized that mobile hybrid applications require extensively external web information, and in most cases, the lack of connectivity in the device is not well managed. These scenarios were common, and caused the aforementioned errors.

***RQ*$_3$ Is RIP suitable to detect crashes and bugs in Android apps?**

RIP is able to detect WEB and HTTP crashes from hybrid applications based on the WebView Javascript console whereas native based tools only detect crashes such as `IOException` or `OutOfMemoryError`. RIP also detects native crashes reported by the system in the `logcat`. This combination of web information and native informations gives **RIP** and advantage over existing tools.

Fig. 4.8: Comparison of a hybrid app exploration between RIP and Firebase Test Lab Robo. Firebase only found 2 states of the app: the main screen and the splash image

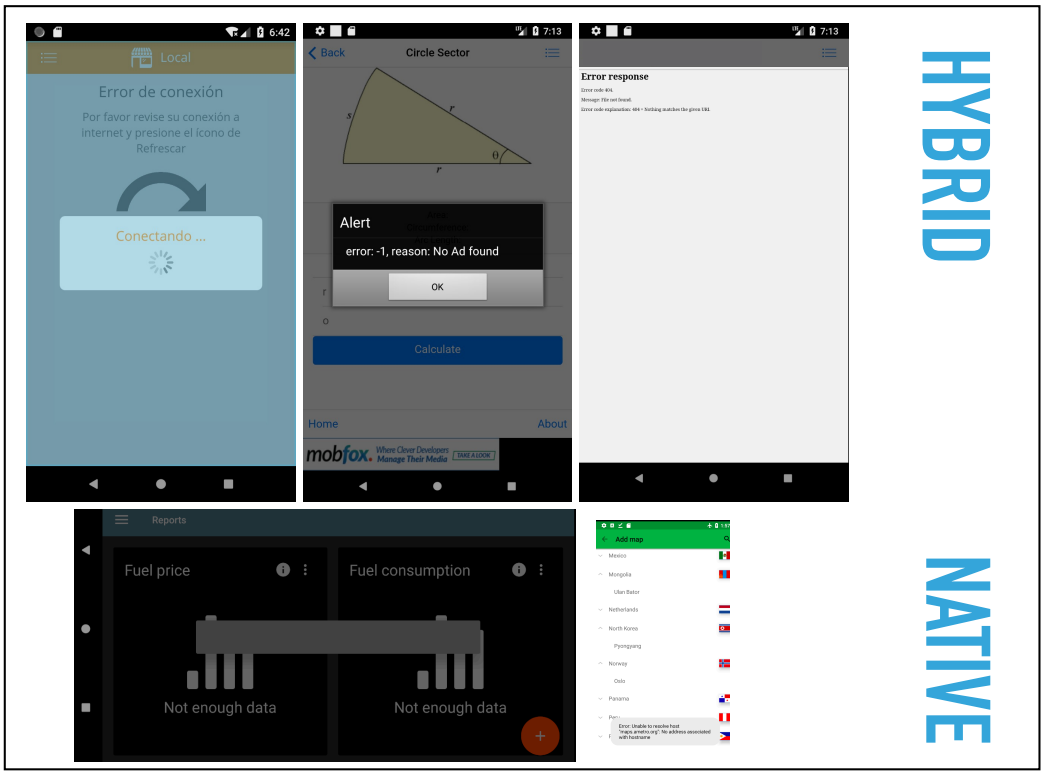

Fig. 4.9: Crashes found by **RIP**

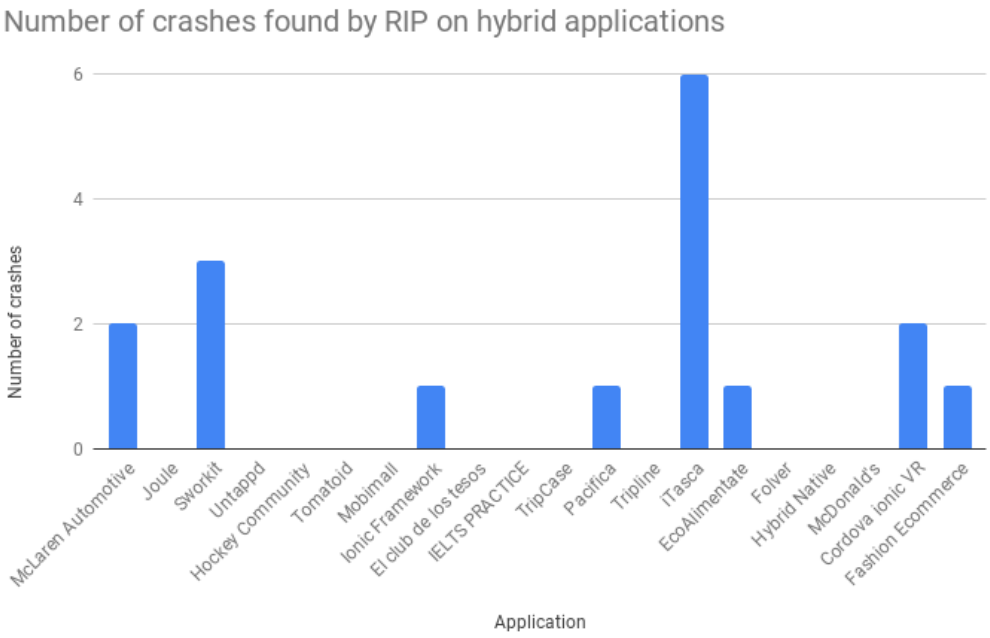

Fig. 4.10: Number of crashes identified by RIP for each hybrid application

.

5 Conclusion

The results from our empirical study suggest that automatically extracting augmented models from Android apps enables better understanding of the apps. For modern mobile applications, ripping apps — but based only on GUI exploration— is not enough because they are context-aware. To that end, context, GUI, usage and domain models should be extracted and combined together to build more useful and comprehensive augmented models.

Still performance of **RIP** exploring native applications is lower than existing baseline tools' (based only on GUI ripping), however enabling contextual changes in exploration allows **RIP** to find states that these tools are not able to detect.

Our tool bridges the gap of ripping hybrid apps, and it is the first approach that focuses in exploring dynamically these applications. **RIP** outperforms industry and academy state of the art tools in the exploration of hybrid apps.

RIP is able to detect WEB and HTTP crashes from native hybrid applications based on the WebView Javascript console whereas native based rippers only detect crashes such as `IOException` or `OutOfMemoryError`. This combination of web information and native informations gives **RIP** and advantage over other existing tools.

The future of mobile application development is uncertain, however, in the short and medium term hybrid applications will start growing faster because of the advances in web technologies, the substantial improvement in the performance of today's mobile devices and the cost reductions of building cross-platform applications with a single language, a single UI and a single technology.

To summarize, the objectives of the thesis were accomplished: an approach for improving automated testing of mobile apps has been developed and integrated into a new tool called **RIP**; a software that extracts multi-models, and performs rip-based crash detection. The performance of this tool has been evaluated and compared, finding its multi-model exploration and ripping capabilities in hybrid apps its main strengths.

5.1 Future work

There is a lot of work to be done regarding tests and experiments, improving our tool and expanding **RIP** to new horizons in mobile software testing.

- Introduce more applications and tools in the empirical study could gives us a better understanding of the possible improvements to RIP and our augmented-model extraction strategy.

- Further studies will be conducted to evaluate and improve automated exploration of the apps, comparing states discovered by our approach against states discovered by users' interactions.

- Improve the GUI ripping algorithm in **RIP** for native apps. **RIP** state discovery strategy based only on the GUI model is below tools like *DroidBot* and *Firebase Test Lab Robo Test*. Improving the ripping strategy in this case, will increment the coverage of states discovered during multi-model ripping.

- Convert **RIP** into a cloud based service. *Firebase Test Lab Robo Test* showed us the benefits of running automated tests in the cloud, freeing testers from manual tasks, enabling parallel execution, and making it more accessible and easy to use.

- Integrate **RIP** with Google Chrome Dev Tools, to analyze hybrid applications DOM without restrictions and inspect more information for these applications.

- Analyze accessibility issues in Android applications based on RIP's state discovery approach. Accessibility services could be discovered and tested to make apps more useful and accessible for everyone.

- In this document, we propose to take advantage of the augmented models and implement multi-model-based testing. Augmented models contain much more information than traditional state diagrams of GUIs. All things considered, multi-models have richer information that will enable generation of more effective test suites. The proposed multi-model could be used to generate test cases, first, using the context variables to define the environmental conditions of the test cases; secondly, generating inputs in test cases by relying on domain entities and attributes; finally, using the GUI and usage information to provide test cases based on developer requirements, such as coverage or specific functionalities. Model based testing strategies could be able to generated rich test suites, containing all the information from the augmented model.

Bibliography

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. “Systematic execution of Android test suites in adverse conditions”. In: *ISSTA'15* (2015) (On page 7).

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. “MobiGUITAR: Automated Model-Based Testing of Mobile Apps”. In: *IEEE Software* 32.5 (2015), pp. 53–59 (On pages 6, 7).

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. “Using GUI ripping for automated testing of Android applications”. In: *ASE'12* (2012) (On pages 6, 13).

[4] J. Bai, W. Wang, Y. Qin, et al. “BridgeTaint: A Bi-Directional Dynamic Taint Tracking Method for JavaScript Bridges in Android Hybrid Applications”. In: *IEEE Transactions on Information Forensics and Security* 14.3 (2019), pp. 677–692 (On page 10).

[5] Alex Baker. *Tasks* (On page 26).

[6] M. Casillo, F. Colace, F. Pascale, S. Lemma, and M. Lombardi. “Context-aware computing for improving the touristic experience: A pervasive app for the Amalfi coast”. In: *M&N'17*. 2017, pp. 1–6 (On page 1).

[7] S. R. Choudhary, A. Gorla, and A. Orso. “Automated Test Input Generation for Android: Are We There Yet? (E)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 429–440 (On pages 5, 27).

[8] IBM Corporation. *Native, web or hybrid mobile-app development*. Tech. rep. IBM Software, 2012 (On pages 7, 20).

[9] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. “Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks”. In: *Proceedings 2014 Network and Distributed System Security Symposium* (2014) (On page 10).

[10] Google. *Android Debug Bridge (adb) | Android Developers*. URL: `https://developer.android.com/studio/command-line/adb` (On page 15).

[11] Google. *Connectivity - Android Developers* (On page 2).

[12] Google. *Firebase Test Lab*. URL: https://firebase.google.com/docs/test-lab/android/overview (On pages 3, 4, 6, 23, 30).

[13] Google. *Layouts - Android Developers*. URL: https://developer.android.com/guide/topics/ui/declaring-layout (On page 21).

[14] Google. *Sensors - Android Developers* (On page 1).

[15] Google. *UI/Application Exerciser Monkey*. URL: https://developer.android.com/studio/test/monkey (On pages 3, 4, 23, 27).

[16] Google. *UI/Application Exerciser Monkey*. URL: https://developers.google.com/web/tools/chrome-devtools/ (On page 9).

[17] Google. *WebView - Android Developers*. URL: https://developer.android.com/reference/android/webkit/WebView (On page 21).

[18] T. Gu, C. Cao, T. Liu, et al. "AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications". In: *ICSME'17*. 2017, pp. 103–114 (On pages 6, 7).

[19] Xing Jin, Xuchao Hu, Kailiang Ying, et al. "Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 66–77 (On page 10).

[20] Xing Jin, Lusha Wang, Tongbo Luo, and Wenliang Du. "Fine-Grained Access Control for HTML5-Based Mobile Applications in Android". In: *Information Security*. Ed. by Yvo Desmedt. Cham: Springer International Publishing, 2015, pp. 309–318 (On page 10).

[21] M. E. Joorabchi, A. Mesbah, and P. Kruchten. "Real Challenges in Mobile App Development". In: *ESEM'13*. 2013, pp. 15–24 (On page 2).

[22] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. "Understanding the Test Automation Culture of App Developers". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015, pp. 1–10 (On page 5).

[23] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. "Understanding the Test Automation Culture of App Developers". In: *ICST'15* (2015) (On page 2).

[24] Jan Kühle. *Car Report* (On page 26).

[25] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. "Automated Extraction of Rich Software Models from Limited System Information". In: *WICSA'16*. 2016, pp. 99–108 (On page 7).

[26] S. Lee, J. Dolby, and S. Ryu. “HybriDroid: Static analysis framework for Android hybrid applications”. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016, pp. 250–261 (On page 10).

[27] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. “DroidBot: a lightweight UI-Guided test input generator for android”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 23–26 (On pages 6, 7, 23, 31).

[28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. *DroidBot Repository* (On page 32).

[29] Chieh-Jan Mike Liang, Ranveer Chandra, Feng Zhao, et al. “Caiipa: Automated large-scale mobile app testing through contextual fuzzing”. In: *MobiCom’14* (2014) (On page 7).

[30] Mario Linares-Vasquez, Kevin Moran, and Denys Poshyvanyk. “Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing”. In: *ICSME’17* (2017) (On pages 2, 5, 6).

[31] Mario Linares-Vasquez, Carlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. “How do Developers Test Android Applications?” In: *ICSME’17* (2017) (On page 2).

[32] Mario Linares-Vasquez, Martin White, Carlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. “Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios”. In: *MSR’15* (2015) (On pages 6, 7, 13).

[33] S. Liñán, L. Bello-Jiménez, M. Arévalo, and M. Linares-Vásquez. “Automated Extraction of Augmented Models for Android Apps”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 549–553 (On page 1).

[34] Ke Mao, Mark Harman, and Yue Jia. “Sapienz: multi-objective automated testing for Android applications”. In: *ISSTA’16* (2016) (On pages 6, 7).

[35] Monaca. *Monaca Debugger A Powerful Solution for HTML5 Hybrid App Testing*. URL: `https://monaca.io/debugger.html` (On page 10).

[36] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Christopher Vendome, and Denys Poshyvanyk. “Automatically Discovering, Reporting and Reproducing Android Application Crashes”. In: *ICST’16* (2016) (On pages 7, 13).

[37] Patrick Mueller. *weinre*. URL: `https://people.apache.org/~pmuellr/weinre/docs/latest/Home.html` (On page 10).

[38] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. “Context Aware Computing for The Internet of Things: A Survey”. In: *IEEE Communications Surveys Tutorials* 16.1 (2014), pp. 414–454 (On page 1).

[39] R. Sharma, P. Soni, K. Shah, and B. Panchal. “Health care application for Android smartphones using Internet of Things (IoT)”. In: *INDIACom’16*. 2016, pp. 1430–1433 (On page 1).

[40] Mohamed Shehab and Abeer AlJarrah. “Reducing Attack Surface on Cordova-based Hybrid Mobile Apps”. In: *Proceedings of the 2Nd International Workshop on Mobile Development Lifecycle*. MobileDeLi ’14. Portland, Oregon, USA: ACM, 2014, pp. 1–8 (On page 10).

[41] thuryn1@gmail.com. *Your local weather* (On page 26).

[42] Simple Mobile Tools. *Simple Calendar* (On page 26).

[43] Guliz Seray Tuncay, Soteris Demetriou, and Carl A. Gunter. “Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 104–115 (On page 10).

[44] S. Yang, H. Zhang, H. Wu, et al. “Static Window Transition Graphs for Android (T)”. In: *ASE’15*. 2015, pp. 658–668 (On pages 16, 19).

[45] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. “Static Control-Flow Analysis of User-Driven Callbacks in Android Applications”. In: *ICSE’15* (2015) (On page 16).

[46] Samer Zein, Norsaremah Salleh, and John Grundy. “A systematic mapping study of mobile application testing techniques”. In: *Journal of Systems and Software* 117 (2016), pp. 334–356 (On page 5).

[47] Chaoshun Zuo, Jianliang Wu, and Shanqing Guo. “Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps”. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS ’15. Singapore, Republic of Singapore: ACM, 2015, pp. 591–596 (On page 10).

List of Figures

List of Tables

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. Firebase is a trademark of Google LLC.