UNIVERSITÄT
DES
SAARLANDES

Learning the Language of Apps

by

Nataniel Pereira Borges Junior

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken
2020

Learning the Language of Apps
Dissertation from Nataniel Pereira Borges Junior
Saarbrücken
2020

Abstract

To explore the functionality of an app, automated test generators systematically identify and interact with its user interface (UI) elements. A key challenge is to synthesize inputs which effectively and efficiently cover app behavior. To do so, a test generator has to choose which elements to interact with but, which interactions to do on each element and which input values to type. In summary, to better test apps, a test generator should know the app's *language*, that is, the language of its *graphical* interactions and the language of its *textual* inputs. In this work, we show how a test generator can learn the language of apps and how this knowledge is modeled to create tests.
We demonstrate how to learn the language of the graphical input prior to testing by combining machine learning and static analysis, and how to refine this knowledge during testing using reinforcement learning. In our experiments, statically learned models resulted in 50% less ineffective actions an average increase in test (code) coverage of 19%, while refining these through reinforcement learning resulted in an additional test (code) coverage of up to 20% . We learn the language of textual inputs, by identifying the semantics of input fields in the UI and querying the web for real-world values. In our experiments, real-world values increase test (code) coverage by $\approx$10%;
Finally, we show how to use context-free grammars to integrate both languages into a single representation (UI grammar), giving back control to the user. This representation can then be: mined from existing tests, associated to the app source code, and used to produce new tests. 82% test cases produced by fuzzing our UI grammar can reach a UI element within the app and 70% of them can reach a specific code location.

Zusammenfassung

Automatisierte Testgeneratoren identifizieren systematisch Elemente der Benutzeroberfläche und interagieren mit ihnen, um die Funktionalität einer App zu erkunden. Eine wichtige Herausforderung besteht darin, Eingaben zu synthetisieren, die das App-Verhalten effektiv und effizient abdecken. Dazu muss ein Testgenerator auswählen, mit welchen Elementen interagiert werden soll, welche Interaktionen jedoch für jedes Element ausgeführt werden sollen und welche Eingabewerte eingegeben werden sollen. Um Apps besser testen zu können, sollte ein Testgenerator die Sprache der App kennen, dh die Sprache ihrer grafischen Interaktionen und die Sprache ihrer Texteingaben. In dieser Arbeit zeigen wir, wie ein Testgenerator die Sprache von Apps lernen kann und wie dieses Wissen modelliert wird, um Tests zu erstellen.
Wir zeigen, wie die Sprache der grafischen Eingabe lernen vor dem Testen durch maschinelles Lernen und statische Analyse kombiniert und wie dieses Wissen weiter verfeinern beim Testen Verstärkung Lernen verwenden. In unseren Experimenten führten statisch erlernte Modelle zu 50% weniger ineffektiven Aktionen, was einer durchschnittlichen Erhöhung der Testabdeckung (Code) von 19% entspricht, während die Verfeinerung dieser durch verstärkendes Lernen zu einer zusätzlichen Testabdeckung (Code) von bis zu 20% führte. Wir lernen die Sprache der Texteingaben, indem wir die Semantik der Eingabefelder in der Benutzeroberfläche identifizieren und das Web nach realen Werten abfragen. In unseren Experimenten erhöhen reale Werte die Testabdeckung (Code) um ca. 10%;
Schließlich zeigen wir, wie kontextfreien Grammatiken verwenden beide Sprachen in einer einzigen Darstellung (UI Grammatik) zu integrieren, wieder die Kontrolle an den Benutzer zu geben. Diese Darstellung kann dann: aus vorhandenen Tests gewonnen, dem App-Quellcode zugeordnet und zur Erstellung neuer Tests verwendet werden. 82% Testfälle, die durch Fuzzing unserer UI-Grammatik erstellt wurden, können ein UI-Element in der App erreichen, und 70% von ihnen können einen bestimmten Code-Speicherort erreichen.

Acknowledgments

This thesis would not have been possible without support from many people.

First and foremost, I want to thank my advisor Andreas, who put a lot of trust in me throughout the whole time in the chair. Thank you, Andreas, for providing such a great working environment and for giving me ample freedom to pursue my research. None of this would have been possible without your support. I'm also grateful to my reviewers Profs. Sven Apel and Mauro Pezzé for the insightful feedback, and Christian Rossow, for taking the time to chair my colloquium.

I would also like to thank Nicolas Pereira Borges (Air Force Technological Institute, Brazil) and Jorge Destri Junior (LabTrans, Brazil) for reading and commenting on drafts of this thesis. You were a great help for pointing out incomprehensible parts, spelling mistakes, and other problems that were present in earlier versions of this thesis.

I am also thankful for all the discussions with my office colleagues, be it during lunch and coffee times at the university or at the retreats in Dagstuhl. You were always available to bounce ideas off, and your feedback was highly appreciated. Special thanks to Jenny Rau, whom I shared the office during the whole Phd and spent countless hours debating ideas.

Finally, I am deeply grateful for all the support I received from my family throughout my life. Your put me in the position to successfully complete this thesis. Thank you for all the effort it took you to get me to where I am today. Last but not least, I want to thank Younjung Jung for her love, her patience and her endless support throughout the last years.

Contents

Chapter 1

Introduction

Smartphones are part of our daily lives. With over 360 million units sold only in 2019 [1], they have become indispensable for both private and professional activities, being used by most people on a daily basis. People use smartphones to perform a wide variety of tasks, from communicating with others—by audio, video, or text—to navigating through the city or searching for information online. With 75% of the worldwide market and an active user-base of more than 2 billion monthly users, Android is the most popular operating system for smartphones [2].

Smartphones are popular because of the many applications (apps) that can be easily downloaded and installed. From banking to social networks, there are apps to assist most of our daily activities. As of August 2019, the official Android app store, Google Play Store, distributes over 2.5 million applications [3]. Although one may assume that the quality of an app is vital to stay competitive in such a market, recent researches indicate that poorly-tested, error-prone apps still exist [4, 5].

Testing is a tried method to increase the quality of an app [6]. Ideally, at least functional testing should be an integral part of an app's development process. Tests can be *written and executed manually*, which is laborious, biased, and time-consuming; *be written manually and executed automatically*, which retains the test bias and high maintenance cost but significantly decreases its execution time; or *be generated and executed automatically*, which is an active research task.

To manually test apps is unfeasible due to their short release cycles, with some major apps releasing a new version at least once per week [7]. Maintaining manually written tests in such quick releases is also a challenge, especially given the lack of operating system support. Each Android smartphone manufacturer has its own set of hardware features and modified operating system versions, which frequently require different test scripts [8]. Due to this particular scenario, in recent years, there was considerable interest in research related to automated test generation for Android apps [9].

While automated testing itself is not a new research topic, testing Android apps is a particularly challenging task. First, Android apps are different than most other software as they are not standalone applications. Instead, they behave as plug-ins to the Android framework, i.e., there is not a single point of entry from which the app execution starts, but instead, they subscribe to the framework which invokes them. Second, most control-flow interactions in Android are governed by specific event-based mechanisms such as the Inter-Component Communication [10], which are handled by the operating system, making them harder to analyze statically. Third, tests must cover different hardware (e.g., screen sizes, sensors) and multiple flavors of the operating system to ensure that the app functionality works correctly. Finally, the widespread use of event-driven framework libraries pose significant obstacles for the systematic generation of test cases [11].

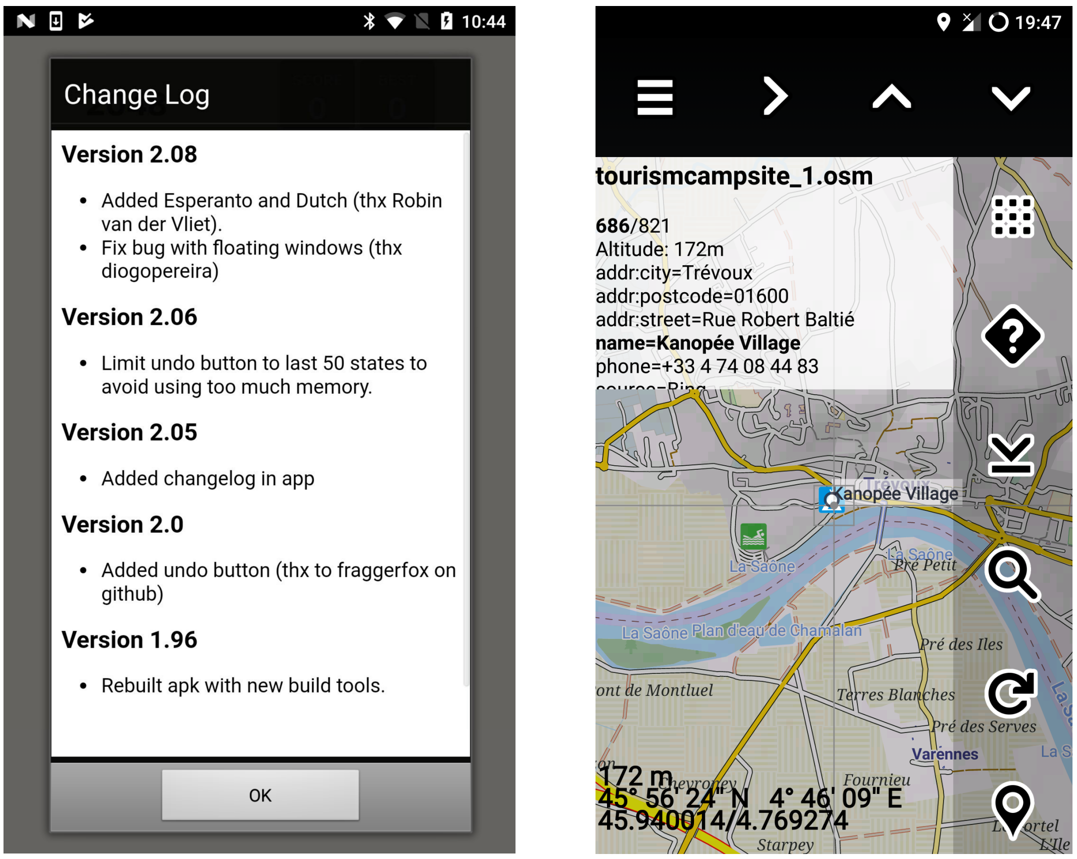
Automated test generators systematically identify and interact with user interface (UI) elements to explore the functionality of a mobile app (including its errors). One key challenge is to synthesize inputs which effectively and efficiently cover app behavior. In order to do so, a test generator must choose the following:

Which UI element to interact with? Most elements visible on a user interface are structuring and do not respond to interactions at all.

How to interact with the UI element? The actionable elements expect specific interactions, such as clicks, swipes, or textual inputs.

Which input values to type? Each input field accepts a specific set of values or patterns.

These are nontrivial challenges, as a test generator not only has to infer the set of user interface elements to interact but also which interactions should be done in each element, to the point of determining which input values have to be typed.



Change Log

Version 2.08
- Added Esperanto and Dutch (thx Robin van der Vliet).
- Fix bug with floating windows (thx diogopereira)

Version 2.06
- Limit undo button to last 50 states to avoid using too much memory.

Version 2.05
- Added changelog in app

Version 2.0
- Added undo button (thx to fraggerfox on github)

Version 1.96
- Rebuilt apk with new build tools.

OK

tourismcampsite_1.osm
686/821
Altitude: 172m
addr:city=Trévoux
addr:postcode=01600
addr:street=Rue Robert Baltié
name=Kanopée Village
phone=+33 4 74 08 44 83

172 m
45° 56' 24" N 4° 46' 09" E
45.940014/4.769274

(a) Change log screen from the app 2048 Puzzle Game[1]

(b) Map screen from the activity tracking app AAT[2]

Figure 1: User interfaces from two apps with different interaction possibilities.

Consider, for example, the apps from Figure 1. It is typically easy for humans to identify active elements since user interfaces follow conventions that humans would learn over time. For a test generator, though, nothing of this is obvious. In Figure 1a, the labels describing individual changes might respond to clicks; the list might be swiped horizontally; the "Change Log" title at the top might be active; and who knows what happens if one swipes across the OK button.

To interact with Figure 1b is challenging, even for humans. What exactly do the individual icons do? Is there further functionality that is not displayed on the screen? Can the map be zoomed in? A human could notice that the last line in the white information box ("tourism-campsite") is not entirely displayed, which indicates that there is more content on it. Indeed this text area is scrollable; swiping on it scrolls the text, eventually revealing buttons at the bottom, which opens up further functionality. A test generator without prior knowledge may click on random parts of the screen, but randomly generating a series of swipes is unlikely to scroll the entire text and reach this new functionality.

How do users know how to interact with apps? They learn general patterns based on apps they previously used. They learn patterns specific to an app while using it. They understand the UI semantics to know which information to enter on the UI. In summary, know the *language of apps*, i.e., they know:

- **the language of graphical inputs**: they know how to generate effective sequences of UI actions, such as clicks and swipes;
- **the language of textual inputs**: they know how to enter correct input values on textual fields.

Similarly, to test apps effectively, a test generator must also be able to learn and consume both *languages*. It typically represents the *language of graphical inputs* as a finite state machine, associating UI action with UI state transitions. *The language of textual inputs*, however, typically comes in the form of regular expressions or context-free grammars [12]. Nevertheless, even if a test generator uses both languages, it faces a second challenge, specific to Android. There are multiple flavors of the operating system, running on different hardware, which may behave differently. Ideally, all variations should be tested, but even efficient UI tests may be expensive to execute. Finally, when a test generator overcomes these challenges, it may still not be able to reach all—or deep—app functionality on its own. Some functionality may be locked behind *gate UIs* [13], which may only be accessible after the user performs complex interactions—sometimes involving multiple apps—such as registering, confirm the registration, and re-opening the app to log in. Thus, it is vital to keep the user in control of the tests by providing it with ways to adapt, extend, and guide them towards its goals.

1.1 Thesis Statement

We split our approach into three parts: *testing*, *learning*, and *modeling*.

[1]`https://f-droid.org/packages/com.uberspot.a2048/`

[2]`https://f-droid.org/de/packages/ch.bailu.aat/`

In *testing*, we lay the base for our approaches. We aim to construct an automated test generator that is easy to extend and provides the standard functionality necessary to our learning and modeling approaches. To be useful in practice, the tool must be able to execute real-world apps from major app stores, form different categories, and with varied sizes. Moreover, it should not require any modification to the operating system as such changes are unfeasible to deploy on a large scale. Finally, the tool must not rely on any static app analysis when generating the tests, as this would prevent the approach from being directly used on native or web-based apps.

During *learning*, we aim to learn how to interact with user interfaces on Android apps and to use this information to aid automated test generation. In this part, we aim to extend our test generator to learn how to interact with apps, and we use this information when generating new tests, resulting in increased functionality coverage. We start by *learning the language of graphical inputs*, we learn how to determine which widgets to interact with statically, and then we proceed to refine this knowledge dynamically. Finally, we *learn the language of textual inputs* by querying the web for real-world inputs.

In *modeling*, we aim to provide a representation that integrates user interface interactions, transitions, and textual input values into a single model. This representation can then be easily read and extended by users according to their needs. It can also be used as a source to produce new, more efficient tests. In this part, we aim to model Android tests as a context-free grammar—a well-known textual-based formalism which the user can easily edit. Moreover, we aim to develop a suite of tools that use such grammar to derive new tests that cover app functionality more efficiently than the original test. The grammar should model possible user interactions, such as clicks and swipes, transitions between UI states, and textual inputs. To be useful in practice, the model should use context-free grammar *as is*, i.e., without introducing any extension. Finally, the model components should be associated with source code, allowing it to produce tests that target both UI elements and code segments.

Note that we do not evaluate the functionality reached by a test in this thesis, i.e., we do not decide if such functionality is critical to the app usage or if it is a side feature. We also do not reason if the functionality worked according to its requirements or not. We acknowledge that such decisions are important, but they require the use of oracles, which we leave as a separate research question.

Finally, the system-specific parts of this thesis are centered around Android. We developed the system-specific parts of this thesis for Android because of its availability as an open-source project, meaning that it is readily available for inspection; and its significant market share. Other operating systems such as iOS and Windows Phone instead require a great deal of reverse engineering effort for understanding their inner workings. Nevertheless, the approaches which we present in this work can be applied not only on Android but also on other mobile operating systems and even in different domains, such as web testing.

1.2 Contributions

This thesis is a step toward improving app quality by enhancing automated test generation capabilities. We present a set of techniques aiming towards this goal as well as infrastructure components that make the app testing itself feasible. In summary, this thesis presents the following original contributions:

- We design and implement an extensible platform for Android test generation. Our platform's design gives developers and researchers ways to quickly implement and combine their custom testing strategies, while abstracting all Android specifics, such as app setup or device communication, away. The platform is presented in Chapter 3.
- We introduce the concept of a *UI interaction model*, which determines how likely a widget is to react to an action. Our model is statically mined from a crowd of apps and is reused to test multiple apps. Building a model based on a large crowd of popular apps captures typical ways of interacting with Android apps. We show how to extend existing test generators to use knowledge mined from a crowd of apps (*interaction model*), guiding test generation towards likely successful UI elements and interactions. The interaction model and how to use it during automated testing are presented in Chapter 4.
- We introduce test generation as an instance of the multi-armed bandit problem (*MAB problem*) and formulate how reinforcement learning, without prior knowledge, can be used to produce test inputs. To the best of our knowledge, this is the first time the *MAB problem* is used as a model for test generation. We show how to enhance our reinforcement learning techniques with statically trained models (*interaction model*) to initialize pre-approximated probability distributions into the models. Our use of reinforcement learning to address test generation as an instance of the *MAB problem* is presented in Chapter 5.
- We extend Link [14], an input querying approach, to obtain input values that are syntactically and semantically valid for testing. Our adaptation of Link is presented in Chapter 6.
- We introduce the concept of UI grammars in which we model app UI languages in the form of a context-free grammar. Additionally, we present how to: automatically mine UI grammars, test with UI grammars, and leverage the association of grammar elements with code locations for code guided testing. Our use of UI grammar is presented in Chapter 7.

1.3 Dissertation Outline

This dissertation's goal is to show how information about UI elements acquired statically and dynamically can be used to guide Android test generation more effectively and whether such tests could be modeled such that even values entered on input fields could be determined. The remainder of this thesis is structured in three parts:

In Part I, we explain the general concepts on which our work is based. This includes Android and GUI testing concepts, such as how Android app user interfaces are structured and existing commercial tools and research prototypes to test apps (Chapter 2). We explain how environment fragmentation, as well as operating system restrictions, affect automated testing on Android. Then, in Chapter 3, we present DM-2, our extensible test generation platform, which we used to implement our techniques. We present each of the components of DM-2 as well as its out-of-the-box capabilities and main extension points.

We then proceed to Part II, where we present how we *learn* the language of apps. Chapter 4 presents the platform-independent probabilistic model (*interaction model*) to use statically gathered data to determine during testing which UI elements are more likely to be actionable, allowing knowledge to be transferred between different apps. We describe each of the components of our model in detail, including its features and limitations. We also describe how to integrate our model with different testing tools, including our DM-2 platform.

Afterward, Chapter 5 presents our use of reinforcement learning to, during testing, learn which UI elements can be interacted with and which type of interaction they are more likely to support. We frame test generation as an instance of the *MAB problem* and show how its traditional techniques perform. Moreover, we also show how such dynamic models can exploit our static *interaction model* to reuse previous knowledge when starting a test and to refine this knowledge while testing.

We conclude Part II by determining how to obtain real input values for textual fields in Chapter 6. We present a technique to query a knowledge base for real-world inputs. We show how this technique can produce syntactically and semantically valid input values. Moreover, we show that such inputs significantly aid test generation.

We then move to the *modeling* part (Part III). In Chapter 7, we introduce the concept of user interface grammars and show how to mine such grammars from test cases, use them to generate new test cases, and to exploit the association between source code and user interface elements to guide test generation towards specific functionality.

We finally conclude our discussion of how to learn the language of apps on Android apps in Chapter 8.

Part I

GUI Testing on Android

Chapter 2

Android and UI Testing

In this chapter, we explain the major concepts necessary to understand the techniques and methods we propose in the remainder of this work. In Section 2.1, we present a brief explanation about how Android apps work, focusing on how apps can structure their user interface and how Android handles events. We also highlight the challenges that arise from fragmentation and present the Android accessibility service, which is the underlying framework mechanism that enables most Android testing. Finally, in Section 2.2, we present existing techniques to test Android apps, including commercially used tools and research prototypes.

2.1 Android

Android is a mobile operating system, primarily designed for touch screen devices. It supports the development of apps, which are installed on devices through app stores. An Android app is commonly composed of their compiled source code, external resources, such as images, and a manifest file.

During installation, the Android OS inspects the app manifest files and register its content and capabilities. Moreover, it uncompresses the apps external resources and source code, and place them predefined locations. Concerning source code, Android apps may incorporate multiple technologies simultaneously. It supports *native* code (e.g., C libraries), that is, elements directly compiled into their final binary form when the app is built. It also supports *interpreted* code (e.g., JavaScript), which is shipped with the app but only loaded and executed at runtime. Finally, it supports *managed* apps, that is, apps built to run on the Android Runtime Environment (ART), a modified version of the Java Virtual Machine (JVM), which translates the app bytecode into the device's binary format.

2.1.1 Activities and Intents

While in most development paradigms, the code of a program starts in a single point of entry, such as the *main* function on C and Java, the same does not hold for Android apps. They instead have multiple entry points, allowing the app to start on different locations. For instance, an email client could start by displaying the received emails when launched without any arguments, but could directly start in an email composition screen when launched through another app.

According to the official documentation[3], to support multiple entry-points Android apps are composed of *Activities*, which are invoked by the operating system. Activities represent the UI where the app can draw its contents. They are also the system components which receive user events, that is, which a user can interact with. In general, each app activity represents one app screen, and, since most apps contain multiple screens, they frequently comprise multiple activities.

Apps, including the operating system launcher, start other apps—or launch a new screen—by invoking their activities. Apps do not, however, start an activity by creating an object. They instead define their *intent* of doing so and request the operating system to create and transition to a new activity.

Intent objects[4] are binders between two activities and represent the intent of an app to do something. They can be manually targeted to a specific activity, or can be targeted towards action types, such as *edit* or insert, and resolved dynamically by the operating system.

2.1.2 UI Structure

Views are a common ancestor to all user interface elements in Android. They represent elements that draw something the user can see and interact with. Each View object occupies a rectangular area on the screen and is responsible for its drawing and event handling. Android widgets[5], such as buttons, labels, and images, are subclasses of View.

ViewGroups are particular types of Views, with a different semantic. Differently from Views, ViewGroups are not meant to be interacted with by the user. They are invisible structuring elements that aggregate *Views* and other *ViewGroups* and determines how they should be positioned on the screen. Android layouts, such as LinearLayout[6] and WebView[7], are subclasses of ViewGroup.

Developers can extend both View and ViewGroup interfaces to create their customized widgets or layouts. Moreover, while the semantics behind the ViewGroups interface means it should be used to create invisible structuring elements, there is no technical limitation that prevents developers from using views to handle user interactions with the UI.

A developer can define views either from code or by describing a tree of views in one or more XML layout files. An Android UI can contain elements defined in multiple layout files, known as *Fragments*, which are automatically inflated when Android renders a window. From a user perspective, an Android UI is always displayed at runtime as a single tree (view hierarchy) composed of all Views.

A *Fragment* represents a behavior or portion of a user interface that can be reused into multiple app screens. They are modular sections of an Activity, with its life cycle, input handling, and which can be added or removed at runtime. In all scenarios, whenever a fragment is loaded into an activity, it is automatically placed under a ViewGroup inside the UI's view hierarchy.

[3]`https://developer.android.com/reference/android/app/Activity`
[4]`https://developer.android.com/reference/android/content/Intent`
[5]`https://developer.android.com/reference/android/widget`
[6]`https://developer.android.com/reference/android/widget/LinearLayout`
[7]`https://developer.android.com/reference/android/webkit/WebView`

2.1.3 User Interaction Event handling

On Android, all interactions with an app happen through *events*. These come in the form of *user interaction events*, such as screen touches or button presses, as well as *system events*, such as received SMS or incoming calls. In this work, we focus exclusively on *user interaction events*, which are forwarded by the OS to the app. Developers can capture the events forwarded by the OS to the app through the *View* interface.

According to Android's official documentation, the View interface supports six types of event listeners[8], namely: onClick, onLongClick, onFocusChange, onKey, onTouch and onCreateContextMenu. These event handling methods are called by the Android framework when the respective action occurs on that object. For example, when a *View*, such as a Button, is clicked, Android invokes the onClickEvent method on the respective object. Moreover, some special events, where the user does not directly interact with the device, are forwarded to the currently focused *View*. An example of such special events are key pressed on an external keyboard.

While any *View* can capture and process events, this is not always the case in real scenarios. Many UI elements are used to display information and are not expected to respond to interactions. *TextViews*, for example, typically display information and seldom have events attached. Similarly, *layouts* are used to structure information and, in theory, should not be interacted with.

Android natively allows developers to capture its current UI state for testing using its native *UiAutomation*[9] service. One can think of the *UiAutomation* as a special type of accessibility service that does not monitor an app's life cycle but provides an API to inspect the UI content and to simulate user actions.

The *UiAutomation*, however, possesses only a limited amount of information about the screen elements. Being a special type of accessibility service, the UIAutomation does not have access to the app's inner workings. The goal of an accessibility service is to aid a user in using apps. For an accessibility service, it is not necessary to know, for example, which types of interaction (event handlers) a *View* support. The user who is consuming the accessibility service is responsible for understanding the app UI and interacting accordingly.

Because of this limitation, the UIAutomation knows only the current View hierarchy, and, for each View, it provides only content (textual and graphical) and state (visible, enabled, and interact-able) information. The associations between UI elements and their implemented event handlers are not available without modifying the Android framework or analyzing the source code of the app. This limitation presents a considerable challenge for efficiently testing apps as the test generator should replace the user and determine how to interact with the app based on the current UI contents.

2.1.4 Accessibility Service

Automated test generation and accessibility share several characteristics. Accessibility service is the mechanism provided by the Android framework to assist users with disabilities in using their devices. It enables the development of services (hence accessibility app) that run in the

[8]`https://developer.android.com/guide/topics/ui/ui-events`
[9]`https://developer.android.com/reference/android/app/UiAutomation`

background and are notified by the operating system when specific events occur, such as when a UI transition occurs, or a new widget has focus. In addition to monitoring events, the accessibility service can be used to determine the content of a screen and interact with any element on it.

From an accessibility perspective: to *use* an app, an *accessibility app* must identify the device screen's content and perform the interaction chosen by the *user*. Similarly, from an automated test generation perspective: to *test* an app, a *test generator* must identify the device screen's content and perform the interaction chosen by its *internal algorithm*.

While test generators do not actively connect to the accessibility service, most rely, at some level, on Android's native UiAutomation [15], which itself is a wrapper around the accessibility service.

2.1.5 Fragmentation

Fragmentation is a phenomenon that refers to the availability of different versions of the system. It occurs either in the *device* or in the *operating system* and, as shown by recent research [16], causes significant compatibility issues for apps. *Device fragmentation* means that there are several different devices with diverse technical specifications, produced by different manufacturers. While this characteristic allowed Android to be used on cheaper devices, with lower hardware specifications, as well as on flagship phones, using cutting-edge hardware, it created an uneven environment where app behavior may change depending on the used environment.

Operating system fragmentation means that an Android device executes one of the 17 major open source releases of the operating system or any modified version of them. Also, although all Android devices share a common code-base, manufacturers develop their customizations to differentiate themselves from their competitors. Such modifications vary from customized application launcher and screen transition animations to optimized modules for battery and memory management. Besides manufacturers, Android has community-supported forks of the operating system, such as Cyanogen [17] and LineageOS [18], which also have their customizations.

Ideally, a test should be able to run in as many devices and operating system versions as possible, mitigating the work on a developer. Thus, it should operate on a higher level of abstraction—states and widgets instead of coordinates due to different screen sizes, for example—and relying on mandatory Android features, such as the accessibility service, instead of on custom operating system instrumentation.

2.2 Existing Techniques to Test Android Apps

There are already various techniques available to test Android apps. While tests may run on the development computer, such as unit tests, in this work, we focus exclusively on UI tests, which require an Android device or emulator to be executed. UI testing is the process of inspecting the app by interacting with its graphical user interface to ensure that users do not encounter unexpected results or have a poor experience. Some of these are industry tools, shipped or not with the operating system; others have been proposed in academic research papers. In this section, we present conventional approaches used to test Android apps. We show that these techniques are either limited in the scenarios that they can be used or that they can be enhanced by knowledge about how to interact with Android apps. We split this section according to the degree of automation provided by the tools into *manually written* and *automatically generated* tests.

2.2.1 Manually Written Tests

Manually written tests allow testing specific app behavior or functionality. They are similar to unit tests; however, they run on Android devices and emulators instead of using the developer's local virtual machine. Such tests are mainly used to unit test functionality, which is either device-dependent or that cannot be easily mocked on regular unit tests.

Espresso [19] is Google's recommended framework to write Android UI tests. While it can be used for black-box testing, many of its functionality can only be effectively used when the source code is available. The main advantage of Espresso is its native UI synchronization mechanism. Before interacting with the app, Espresso automatically waits until the app stabilizes by monitoring its message queue, the list of asynchronous tasks being executed, and by checking that resources are idle. This native synchronization mechanism increases the test reproducibility.

UI Automator [15] is Google's framework to write cross-app functional UI testing. It relies on a mix between Android's internal automation engine and accessibility service to interact with UI elements. Differently from Espresso, UI Automator does not require an app's source code or internal implementation details. Moreover, it is unable to determine when the app's user interface has stabilized, as Espresso does, leaving the developer responsible for this verification.

MonkeyRunner [20] is a Google API to write Android tests. It does not rely on source code and can be used for black-box testing. Under-the-hood, MonkeyRunner uses UI Automator to interact with the app but provides a user-friendly API for developers. It can interact with an app by issuing keystrokes to screen coordinates. It also allows developers to control some of the device functions such as sleep, wake, reboot, and take a screenshot However, it does not know the user interface elements and can only interact with them through their coordinates. For more efficient testing, it should be combined with other techniques that detect the UI elements on the screen.

Robotium [21] is a popular open-source framework for writing gray-box UI tests for Android. Its main advantages are to allow the developer to manipulate the app's life cycle during testing, such as to create an activity and immediately terminate it; and to start a test in any app activity, without following the typical app workflow.

Appium [22] is a popular cross-platform testing framework which also supports Android. Similar to MonkeyRunner, it does not require the app source code to be available. Moreover, it also relies on UI Automator to interact with an app and provides the developers with a more user-friendly API. While it does not offer new features over MonkeyRunner, Appium's main advantage is to provide a single interface so that the same test can be executed, for example, on Android and iOS versions of the same app.

Robolectric [23] takes a different approach to Android testing. It is an open-source framework for running Android tests outside of the Android emulator and on top of the local Java Virtual Machine (JVM). In this approach, a developer uses shadow classes that simulate their Android counterparts' behavior and override the calls to them through reflection. Robolectric goal is to speed up testing on Android apps significantly; however, only testing the app on mock classes may not fully reflect the behavior of the operating system. Thus, Robolectric should not be used alone, but alongside another testing approach.

Instead of relying on test scripts, Reram [24] follows a record and replay approach to represent complicated non-discrete gestures, such as those frequently used in games. Reram monitors the input event buffer from the device (`/dev/input/event*`) while the app is used. It then can be used to replay the same events, at the same time interval, when re-testing the app. Similarly, Valera [25, 26] also works through recording and replaying app traces. However, it can replay not only user actions but also system events such as sensor data and network responses.

Finally, Puma [27] is a framework that encapsulates testing abstractions and exposes them to the developer through its event-driven scripting language. It splits the test logic from the app interaction, allowing the same test to be used in different platforms. Moreover, by isolating the app interaction mechanism from the app testing logic, PUMA automatically propagates any improvements in the app interaction mechanics to all tests.

Tools that require the developer to write tests manually have several limitations. First, such tests can only test specific functionality in specific ways. While it is theoretically possible to write distinct tests to cover all existing paths in a program, such a test suite is laborious and expansive to develop and maintain. A second issue with manually written tests is bias. These tests inherit the bias of the developers that wrote them and are seldom reliable tests to evaluate an app's robustness or security [28]. One last issue with manually written tests is that most approaches in this category rely either on available source code (white or gray-box testing) or on stable user identifiers, both of which are not always available. The main advantage of manually written tests is the existence of an oracle (the test developer), which can validate, for each UI state action, if the app has behaved correctly.

2.2.2 Automatically Generated Tests

Automated test generation in mobile apps is an active research field with, according to a 2018 survey [9], more than 100 publications in the past nine years. Test input generation techniques are commonly classified into three categories [29]: *random*, *model-based*, and *exploratory*, according to their strategy to interact with the app.

2.2.2.1 Random strategies

Random strategies generate inputs at random to explore an app's behavior. Many tools are implementing this kind of strategy, often used to test the robustness of apps. Monkey [30] is the most frequently used tool implementing random testing. It is Google's automated random testing tool, which is a part of the Android software development kit. It generates user events such as clicks, touches, or gestures, using a basic random strategy, and system-level events. It is often used to stress-test applications and can generate reports if the app under test crashes or receives non-handled exceptions. Dynodroid [31] also applies random testing, but in a slightly more efficient way than Monkey by taking the context into account when selecting an input. It can also generate system events. To do so, it requires instrumenting the Android framework. It checks which system events are relevant for the app under test by monitoring when the app registers listeners within the Android framework. BBoxTester [32] generates explicitly invalid data as inputs to an app to check its error handling capabilities. Dagger [33] uses MonkeyRunner to randomly interact with the app while tracking provenance relationships and observing the app interactions with the operating system and uses this information to identify malicious apps. DroidMate [34] is a fully automatic GUI execution generator. It works on devices and emulators out of the box, with no root access or modifications to the OS, and can be easily extended, being the tool used as a base for our experiments.

While random approaches are widely used and are useful for testing the robustness of an app, they have a few limitations. They can seldom reach actions which require specific sequences of events, such as login in into an app, and they do not know how to interact with UI elements, resulting in several ineffective actions.

2.2.2.2 Model-based strategies

Another category of exploration strategies is *model-based strategies*. They extract and use a model of the app under test to systematically generate inputs.

Some tools extract the app model dynamically, that is, during app execution. One of the earliest examples of this approach is AndroidRipper [35]—and its follow up work MobiGUITAR [36]—which uses a user-interface driven *ripper* to systematically traverse the app's user interface, dynamically building a state transition model of the app. Another very similar tool is DroidBot [37]. However, while both tools build similar app models, DroidBot, works entirely without app instrumentation, making it particularly useful to examine malware, as malicious apps often validate themselves before triggering malicious behaviors. Another approach to extract app models is taken by SwiftHand [38]. Instead of learning the app model in a single run, it follows an interactive approach. It first learns an initial model of the app during testing. It then consumes this model to generate inputs, execute them and uses their results to refine the initial model. One last dynamic approach to extract a model for testing is taken by iMPAcT [39]. Instead of extracting a model of the app itself, iMPAcT focuses on each app screen. It then compares each screen against its catalog of patterns and uses them to determine how to test the UI better.

Some tools rely exclusively on static analysis to extract the model of an app. Among such tools is A^3E Targeted [40]. It uses a static data flow analysis to build an activity transition graph that captures possible transitions among activities. It then systematically explores the graph, directing the exploration to cover all activities, especially those deep within the app, and would be difficult to reach during regular use. TrimDroid [41] also extracts models of the app statically. It extracts an interface model, containing widgets and events for each Activity, and an activity transition model, determining the relationships between activities. It then uses these models to reduce the number of combinatorics needed to test the app UI.

Finally, some tools rely on both dynamic and static analysis to extract an app model. ORBIT [42], for example, uses static analysis to determine the set of events supported by each app activity. Then, it reverse-engineers a model of the app by systematically exercising these events. Similarly, *SmartDroid* [43] exploits static analysis to generate activities and function call graphs to identify paths that should be explored. One of the newest Android test generator, Stoat [44], also employs both static and dynamic analysis to reverse-engineer a stochastic model of the app's GUI. It then iteratively mutate and refine the stochastic model using an adapted version of Gibbs sampling, guiding test generation towards model and code coverage.

While model-based approaches are a significant part of the available testing techniques, they also have limitations. Model-based techniques that rely on static analysis produce inherently incomplete models, as some app behaviors can only be seen dynamically. Moreover, they can be challenging to apply on industry-size apps, as they frequently combine different technologies, such as Java, JavaScript, and C code, within the same app. Model-based techniques that do not rely on static analysis face similar challenges as random testing techniques as they perform several ineffective actions during exploration.

2.2.2.3 Exploratory Strategies

The third main category of automated testing techniques is *exploratory strategies*. Tools in this category employ a wide variety of techniques to guide test generation towards specific targets or increasing code coverage.

Some approaches use symbolic execution or concolic testing. AppAudit [45] uses them to guide test generation towards specific targets to prune false positives from static analysis. IntelliDroid [46] uses them to generate feasible event sequences that trigger malicious app behavior. Finally, *ACTEve* [47] uses them to trigger as much app behavior as possible.

Similarly, other techniques are also used with different goals. *EvoDroid* [48] and *Sapienz* [49] use search-based algorithms—evolutionary or combined with random fuzzing—to improve test coverage. CuriousDroid [50] and FraudDroid [51] decompose the application UI on-the-fly, creating context-based models tailored to the current user layout. While CuriousDroid is focused on extracting dynamic sandboxes, FraudDroid guides the exploration towards potentially fraudulent ads. A^3E Depth-First [40] uses the same static activity transition graphs as A^3E Targeted. However, it explores activities and UI elements in a depth-first manner, traversing the app slower, but more systematically than A^3E Targeted. Similarly, CrashScope [52] also builds a transition graph and explores the app in a depth-first manner. It employs a more advanced textual input generator with the goal of not only finding crashes but also ensuring the reproducibility of its test cases.

Exploratory testing techniques have similar limitations to model-based ones. When using static analysis, they rely on an inherently incomplete app model. They also face challenges over complex apps, which use multiple technologies, or whose code is mostly executed on a server instead of the device (web apps). When not relying on static analysis, exploratory techniques perform several ineffective actions, similar to random strategies. Finally, when faced with unknown states or with multiple equally good alternatives, exploratory strategies by default fall-back to a random selection.

Chapter **3**

A Platform for Android Test Generation

This chapter is taken, directly or with minor modifications, from our 2018 ASE paper *DroidMate-2: A Platform for Android Test Generation* [53] and from our 2019 ICST paper *Why Does this App Need this Data? Automatic Tightening of Resource Access* [54]. My contribution in these works is as follows: (I) original idea of an extensible testing platform; (II) idea and development of the exploration strategies; (III) idea and development of the monitoring proxy; (IV) original idea of extensible model features; (V) partial evaluation.

Test generation is a continually evolving subject—as testing techniques improve, so does the complexity of the apps being tested, leading to a never-ending demand for more advanced testing techniques to cover app behavior as effectively and efficiently as possible. While literature presents a large variety of test generators, most tools have one of the following limitations: they rely on *operating system customizations* for more advanced test generation algorithms [55, 46], or are tightly coupled to a single testing strategy [35, 56], or do not have an underlying app model to allow tests to be re-executed on different configurations [30, 20].

The test generation techniques we present later in this work require a test generator that can be easily extended and where different testing strategies can be combined. Moreover, to be able to learn how to use apps and to reapply this knowledge on different apps, the test strategies must operate on abstractions that could be transferred between apps. Finally, to address the challenges of fragmentation, the tool should rely only on core Android components, allowing it to run on different Android versions on devices from different manufacturers. Thus, we designed DM-2, an extensible test generation platform whose details are present in this chapter.

DM-2 is based on the DroidMate project and inherits its main benefits. It can run on any Android version between 6 (API 23) and 9 (API 28) on physical devices and emulators without root privileges or operating system customization. DM-2 nevertheless goes far beyond the functionality provided by DroidMate. While DroidMate was a test input generator with API monitoring capabilities, DM-2 offers easy to use mechanics for developers to implement systematic testing strategies, which can be used individually or combined with others. These strategies are built on top of an internal app model that re-identifies widgets and states during testing, even on different devices and configurations. Finally, DM-2 comes with embedded mechanisms for coverage analysis, crash detection, and test reproducibility.

DM-2 is a distributed system, running partially on the Android device and partially on the host computer. Its architecture comprises three major components, as illustrated by the black frames in Figure 2 and a monitoring proxy (dashed frame). In the remainder of this chapter, we present each of these components. We start by presenting the ones on the host side. In Section 3.1, we present the *app model*, which abstracts all Android related information. We then present which exploration strategies and interaction types are shipped with DM-2 and how developers can create their Section 3.2 (*exploration engine*).

We then proceed to explain the *automation engine* (Section 3.3), which handles all communication with a device and translates the high-level interactions used in the automation engine into accessibility events that can be processed by Android. We then proceed to the device side components by presenting the *monitoring proxy* in Section 3.4, which can intercept and modify calls between the operating system and the app under test. Finally, we present a proof-of-concept evaluation (Section 3.6) comparing DM-2 out-of-the-box against other state-of-the-art test generators, DM-2's limitations (Section 3.7), and we conclude this chapter in Section 3.9.

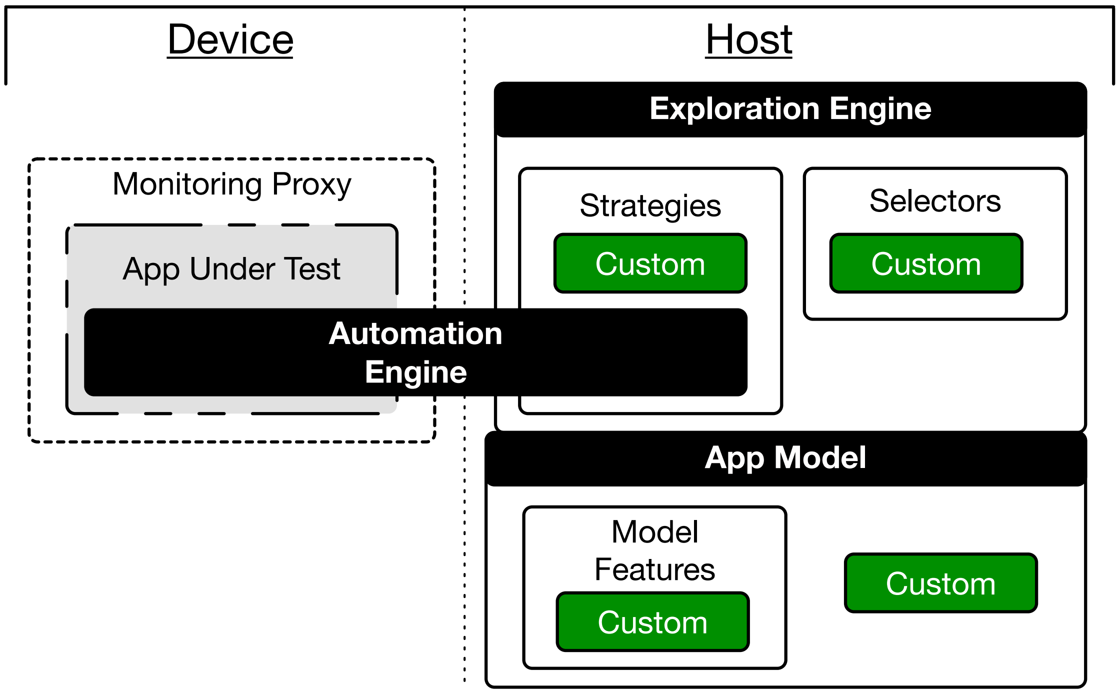


Figure 2: DM-2's conceptual architecture, with elements distributed among the device and the host computer. In green are highlighting the places where the developer can add custom implementations.

3.1 App Model

During exploration, DM-2 builds an internal representation of the app under test composed of *widgets*, *UI states*, and *transitions*. Each UI state encompasses a set of UI elements, some of which are actionable, meaning that the user can, for example, tick, click, or long-click them. This interaction may *transition* the model into another state.

We designed our app model according to the following guidelines:

Element identification: It should identify conceptually identical states, i.e., slight differences in the rendering like the highlighting of previously interacted elements should not be interpreted as a different state. Moreover, widgets should be re-identifiable, even in distinct states.

Abstraction: Its elements should be abstract from the device and operating system in which the test is executed; that is, it should re-identify the same UI elements and state should when executing the app on different devices.

Extensibility: It should be easily extensible, supporting different definitions of UI states and widgets.

3.1.1 Modeling Widgets

Android does not provide a unique identifier for widgets. The closest information it has is a *resource ID*, which is specified by the developer. Nevertheless, the *resource ID* is not only optional and seldom used, but also, can be reused in the same state. Our model, thus, requires a metric to identify widgets. From the Android accessibility service, we obtain 22 properties, listed in Table 1, and we calculate:

visible regions: the widget areas not covered by a child widget, i.e., bounds minus the area covered by its visible children;

visual content: the UI content (screenshot) of the widget, cropped by its visible regions;

fallback ID: the widget position on the UI hierarchy tree produced by the accessibility service, calculated using the parent ID and the index from the current element and all its ancestors until the UI hierarchy root.

Table 1: Widgets properties which can be extracted from the Android Accessibility service

internal id	*text*	hint text	content description
class name	*parent ID*	*index*	package name
is input field	is password	resource ID	input type
checkable	bounds	visible	checked
clickable	enabled	focuseable	focused
scrollable	long-clickable		

These properties can be clustered into stateless (underlined) and stateful (others). Stateless properties characterize the widget itself, such as their name, textual content, and resource ID. Stateful properties are used to determine how the widget is configured at the moment, i.e., enabled, visible, focused. We exploit this distinction to identify a widget as a pair:

$$(id_w,\, cfg_w)$$

where id_w (widget identifier) is a set of stateless properties and cfg_w (widget configuration) is a set of stateful properties. We make this distinction to allow widgets to be re-identified independently of their current configuration and to be re-identified between different devices.

We use the *configuration identifier* only to distinguish between different instances of the same widget; thus, it suffices to define cfg_w as:

$$\begin{aligned} cfg_w = \{ \ & \text{focuseable, focused, scrollable,} \\ & \text{long-clickable, checkable, checked,} \\ & \text{text, visible bounds, visual content} \ \} \end{aligned}$$

Differently from the *configuration identifier*, we cannot use a single definition for all types of widgets. To describe a label, for example, it suffices to use its textual content; however, the same information cannot identify an input field—as the textual content changes with each typed letter—or an image which has no textual content. We, thus produce a id_w according to the following definition:

$$id_w = \begin{cases} \text{hint text, content desc., or resource ID} & \text{if is input field} \\ \text{hint text, text, and content desc.} & \text{if has textual content} \\ \text{resource ID} & \text{if has resource ID} \\ \text{class name, package name, and fallback ID} & \text{otherwise} \end{cases}$$

3.1.2 Modeling States

Similarly to widgets, Android does not provide any reliable way to identify UI states. While one may consider each activity as a state, Android supports a single-activity multiple-fragment architecture, where the developer loads at runtime different fragments in the same activity. Our model, thus, requires a metric to identify UI states.

We opted to define app states from a user perspective based on the widgets available on the screen and their current configuration. To be resilient against external sources, we ignore elements that do not belong to the app under test, such as advertisement containers and other apps launched through intents[10] when determining a UI state. Finally, analogously to the widgets, our app model should be able to distinguish between different UI states and between different configurations of the same state. We, thus, define a UI state as a pair:

$$(id_s,\ cfg_s)$$

where id_s (state identifier) is the set of id_w from the relevant widgets on the screen, i.e., those that have no children or have a textual content or are actionable; and cfg_s (state configuration) is a set of id_w and cfg_w from all widgets. Our state identifier allows states to be re-identified independently of their current configuration, as well as to be re-identified between different devices.

[10]We filter external widgets based on their package name.

3.1.3 Extending the Model

While DM-2 provides an out of the box app model with a metric to identify UI states and widgets, such a heuristic may not be suitable for all apps. To support such a scenario, DM-2's app model was designed to be highly extensible. We cluster DM-2's app model extensions into three categories: *properties*, *ID computation*, and *ad-hoc features*.

Properties. DM-2 uses all properties available in the Android accessibility service, and, as per our guidelines, it does not rely on any device-specific information. While this makes DM-2 usable as a general testing tool, when targeting specific devices or operating system versions, more information may be available. Moreover, the existing widget or state properties can be used to produce more information, such as textual content for image widgets and semantic topics for UI states. For such scenarios, DM-2 allows its internal widget and state definition to be replaced by one written by the developed.

ID Computation. The unique and configuration identifiers from both widget and states from DM-2 aims to provide a balance between specialization, that is, distinguishing each minor UI change as a new widget or state, and generalization, i.e., combining multiple widgets and states into a single one. Such a metric may not be suitable for all scenarios; thus, it is possible to change how DM-2 computes both identifiers easily. For example, on scenarios that rely on textual language content to guide testing, it may be useful to disregard elements without a textual content while determining the state identifier.

Ad-Hoc Features. It is possible to plug-in observers (hence model features) to monitor changes on DM-2's app model without extending the app model itself. Such observers are faster and easier to implement than model extensions and should be used if there are no changes in properties or ID computation. Model features can be used asynchronously, to gather data which can be used for later analysis, or synchronously, computing additional information which can be used by strategies and selectors to determine the next interaction step. DM-2 comes out-of-the-box with model features to track statement coverage, inspect which activities and APIs are triggered, and report which actions and widgets crashed the app during the test.

3.2 Exploration Engine

The exploration engine is responsible for determining how to interact with the app. It consumes the UI states and widgets from the *app model* and uses the *automation engine* to send commands to the device.

DroidMate explored apps through an *exploration loop*. In this loop, before each action, it obtained the current UI state of the app. It then determined which widgets could be interacted with and randomly choose one. This approach is similar to the one used by most other test generators [35, 37]; however, it is hard to extend. Moreover, it assumes that a single exploration algorithm can handle all UI states, leading complex implementations which are difficult to develop and maintain.

We designed DM-2's exploration engine to mitigate such issues. We established the following guidelines for DM-2:

Extensibility It should be easy for developers to create their test algorithms.

Reuse Each test algorithm should be reusable—partially or in full—by other test generators.

Ready-to-Use It should be capable of handling the most common scenarios out-of-the-box.

DM-2 extends DroidMate's exploration loop mechanics with a pool of *strategies* and *selectors*. Strategies model how to interact with the current UI state and selectors determine which strategy to use. DM-2's modified exploration loop is illustrated in Figure 3. Except for the *device*, all components in this figure belong to DM-2. For each action, DM-2 chooses, from its selector pool, which strategy should be used. It then asks the chosen strategy to interact with the app. When a strategy issues a *terminate* action, the exploration loop stops. This two-tier mechanism allows different testing algorithms to be used in different UI states. Moreover, it allows the same algorithm to used in different test generators, providing better *extensibility* and *reuse*.

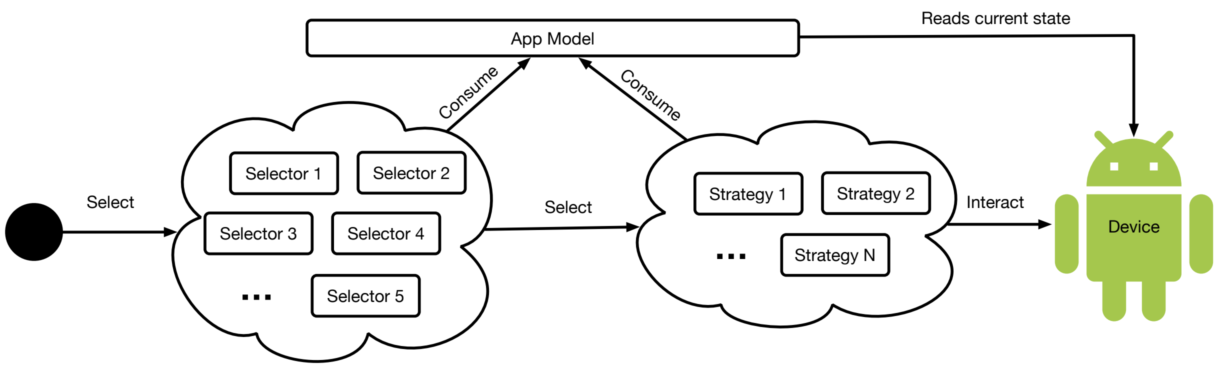


Figure 3: Determining how to interact with an app in DM-2's exploration loop

3.2.1 Interacting With The App

DM-2 models app interactions through *strategies*. Based on the current state of the app model, a strategy decides which exploration actions should be issued next. An interaction with an app can simple, such as *click on coordinates* (x, y), or complex, such as *close the app, enable the device wi-fi, Bluetooth and restart the app from its initial screen*. Formally, strategies are map functions $s(c) \to I$ which receive the current UI state (c) and produce a list of exploration actions (I).

To abstract all device-specific information and mitigate fragmentation-related issues, strategies do not interact directly with the device. Instead, they obtain the current UI state from the *app model* and produce exploration actions that are processed through the automation engine.

While DM-2 allows developers to create their testing strategies, it also comes with a set of strategies to handle common scenarios. Out-of-the-box it comes with the following strategies:

Reset App This strategy resets the phone back to a known state and starts the app to tet. It first ensures that the wi-fi connection on the device is enabled. This step is necessary because most apps rely on an active Internet connection to work correctly. It then closes any open application on the device and puts it back on its home screen. Finally, it opens the app menu, locates the launcher icon associated with the app, clicks on it, and, if the app starts with an open keyboard, closes it. **Limitation:** This strategy assumes that each app has a single launcher activity on the menu. While Android supports multiple launchers for a single app, we have not observed such behavior often during development.

Terminate Test This strategy denotes the end of a test. It ensures that all asynchronous model features are up-to-date and closes the app.

Press Back Press the back button of the device. This strategy can be used to return to the app when another app is displayed without having to restart the whole app. Such an app change happens, for example, when the test generator interacts with an advertisement, triggers a share or send email functionality.

Biased-Random Exploration This strategy randomly selects a UI element from the current screen, among those who have been least explored and click, long-click, or swipe on it. We count how often each app widget was actioned through a model feature to determine the least explored. We compute how many times each widget has been interacted in the current UI state and select those least interacted with. We then filter these widgets based on the number of times they have interacted over all states in the test. If this filter returns more than one widget, we randomly chose one.

Random Exploration This strategy simulates the behavior of MONKEY [30]. It randomly clicks on the coordinate of any UI element on the screen, disregarding if the widget can be interacted with or not.

Allow/Deny Permission These strategies automatically identify if the current UI state is a runtime permission dialog and clicks on the accept or reject button.

PlayBack This strategy selects the interactions from a previously recorded test and attempts to replay it. If the action interacted with a widget (e.g., click, long click), it attempts to re-execute it if the same widget is available on the current state. If the action interacted with the state (e.g., back) it only re-execute it if the app is in the same state.

3.2.2 Choosing Strategies

While strategies model how to interact with the app, *selectors* model which strategy to choose. They can be used to select new UI elements to interact with, to restart the app when it reaches a state that does not belong to the app, attempt to close a randomly displayed advertisement pop-up, or terminate an exploration after a specified time or number of actions. Selectors make it easy for developers to reuse strategies on different scenarios, and to handle exceptional cases efficiently, e.g., *if the current state is login, then invoke the custom login strategy.*

Differently from traditional test generators, DM-2 does not assume that a simple algorithm will suffice to handle possible app interactions. Instead, it uses a set of criteria (pool) to determine the best strategy to handle each UI state. Formally, a selector is defined as a pair $(p, f(c) \to s)$ where p is its priority and $f(c)$ is a mapping function f from a model context c (e.g., the current state or the action trace) to an exploration strategy s. For each selector in its pool, DM-2 calculates if the selection criteria is fulfilled $f(c) \neq \varnothing$ and choses the one with higher priority p.

DM-2 is shipped out-of-the-box with a set of selectors. In Table 2 we present the each of the selectors, as well as their description and which strategy Section 3.2.1 they trigger.

Table 2: Selectors shipped with DM-2

Selector	**Triggers**	**Description and usage scenario**
Time-Based Terminate	Terminate Test	Terminate the exploration after a specific amount of time (e.g. explore the app for 30 minutes). This selector should be used when running a test for a fixed time.
Action-Based Terminate	Terminate Test	Terminate the exploration a specific number of actions (e.g. explore the app for 500 actions). This selector should be used when running a test for a fixed number of actions, for example, when comparing different testing approaches.
Initial Reset	Reset App	Start the app to test. This selector is only triggered in the first test action. After the initial action all app resets should be handled through other selectors.
Reset on Crash	Reset App	Restart the app, if there is an *application not responding* (ANR) dialog on the screen. This selector ensures that, even if the app crashed, the DM-2 will attempt to restart it to continue the exploration until a selector terminates the test.
Reset on Intervals	Reset App	Reset the app after a predefined number of actions (e.g. restart the app every 100 actions). This selector allows the exploration to try new app branches from its initial screen.
Randomly Press Back	Press Back	When active, this selector randomly—with a user-defined probability—presses the back button of the device (e.g. press back with 5% probability).
Handle Google Advertisement	Press Back	When active, this selector handles advertisements from Google's advertisement library (identified by the package *com.android.vending*). If the widgets in the current state belong to the Google's advertisement library, press back to close the ad.
Allow Permission	Allow Permission	Press *allow* on all runtime permission dialogs displayed during the test.
Deny Permission	Deny Permission	Press *deny* on all runtime permission dialogs displayed during the test.
Explore Randomly	Random Exploration	Performs a random action (similar to Monkey).
Default Explore	Biased-Random Exploration	Performs a random action using the biased-random action.

3.3 Automation Engine

The *Automation Engine* acts as a bridge between the exploration strategies and the app under test. It abstracts and manages all Android-related communication using a synchronous protocol based on *actions* and *responses*. It runs simultaneously on the device (server) and the host side (client).

Strategies send lists of exploration actions, such as *click login button* or *type "123" on password field*, to the *Automation Engine* client, which forwards them to its server-side instance and halts the test while this action is processed. The *Automation Engine* server translates exploration actions into accessibility events or API calls, to emulate human activity. To *click login button*, for example, it identifies the widget which represents the login button and sends a click event to a coordinate inside the button's visible region. Similarly, to *type "123" on password field*, it first identifies the respective widget on the screen, clicks on it to activate the keyboard, enters the text, and closes the keyboard.

Before issuing a response to the *Automation Engine* client, the *Automation Engine* server must wait until the app stabilizes, that is, until it finishes performing the previous action and has at least one element from the app to interact with. This synchronization is a natural bottleneck of most Android test generators that allow for state-aware UI actions. However, it is necessary to correctly emulate a user's behavior, who has to wait until a new screen is loaded or animation is complete to continue using the app.

The time to execute an action varies according to the functionality being performed – ticking a checkbox is faster than clicking a login button – and external factors, such as network speed and server availability. The *Automation Engine* copes with varying load times as follows: first, it waits for the device to be idle, that is, ready to receive and handle commands again. It then waits until at least one UI element can be interacted with. It discards any UI elements displayed only during this transition period, e.g., progress bars, as they do not provide any explorable behavior.

Once the app stabilizes, the *Automation Engine* server issues a response containing the structural (screen dump) and visual (screenshot) state of the device to the *Automation Engine* client, which notifies the app model for an update, and the *Exploration Engine* to wait for the next action.

3.4 Monitoring Proxy

The *Monitoring Proxy* is a payload deployed to the device in order to work as a proxy between the app and OS, as shown in Figure 4. It monitors and user-defined resources and intercepts API invocations without changing the app code.

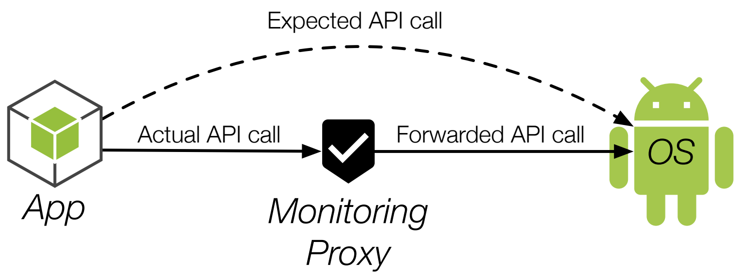


Figure 4: Interaction between the app and the operating system through DM-2's monitoring proxy

Besides monitoring API calls, *Monitoring Proxy* can be used to manipulate their behavior. It allows developers to write their custom code to handle each API invocation, forwarding it or not to the operating system. When enabled, DM-2's *Monitoring Proxy* comes with three configurable security policies for each Android API: *monitor*, *deny*, or *mock*.

Monitor When using this policy, the *Monitoring Proxy* logs each API call, but does not manipulate its behavior. It logs the stack trace before the API call, the received parameters (if any) before forwarding the API to the operating system. It then logs the APIs return value, if any.

Deny When this policy is active, *Monitoring Proxy* prevents the app from accessing the configured APIs. Whenever the API is invoked, *Monitoring Proxy* intercepts the call and does not forward the API call to the operating system, raising a *SecurityException* instead.

Mock Similarly to the *Deny* policy, this policy also prevents the app from accessing the configured APIs. However, after intercepting the API call, this policy does not trigger an exception. Instead, it attempts to return a mocked value to the app. By default, it returns Java's standard initialization values for primitive types, such as 0 for integers and empties for strings.

To intercept method calls without requiring the app or operating system modification, *Monitoring Proxy* relies on Arthook [57], a callee-site in-memory rewriting technique to inject method hooks into an app. While the technique itself can be applied to any system, its implementation relies on Thumb-2 type instructions from ARM processors; thus, *Monitoring Proxy* can only be used on physical devices or emulators with the ARM processor architecture.

3.5 Obtaining Code Coverage

The goal of testing is to find existing bugs; therefore, an automated test generator should ideally be evaluated regarding the bugs it can find. However, except for specially curated datasets [58, 59], it is not possible to know how many—and which—bugs exist in an app.

To avoid such a problem, test generators measure how much app functionality they can test [29]. While it is also not possible to know how much functionality exists in an app, test generators approximate the amount of functionality in an app by their lines of code [60], and such a metric is a good predictor for fault detection [61].

Since DM-2's goal is to allow developers to write, combine, and compare different testing techniques, DM-2 is shipped out-of-the-box with an embedded code coverage measurement mechanism.

DM-2 instruments the app's binary file (APK) to track statement coverage. It uses ApkTool [62] and Soot [63] to repack app and inject coverage tracking statements in its compiled Java code. The statement coverage tracked by DM-2 is relative only to the Java part of the app code, which is present within the app binary. DM-2 is unable to monitor native and JavaScript portions of apps, as well as dynamically loaded statements or external app components, such as a web server.

3.6 Evaluation

While DM-2 is designed to be an extensible test generation platform, it can also be used out-of-the-box as a test generator. In the remainder of this work, we use this DM-2 configuration as the baseline for comparing our techniques, as it provides the same performance baseline. We, thus, conducted a set of experiments on how efficient and effective DM-2 is. In particular, we aim to evaluate: *does* DM-2 *covers app behavior efficiently?*

3.6.1 Experimental Setup

We compared DM-2 against DroidBot, which presents similar functionality and has been shown to outperform most current test generators [64] and Monkey, the standard test generator from Android. We have not compared against other tools as they either:

- require a specific (and old) Android version [35, 36]
- require OS modifications [47, 46]
- were outperformed by Monkey or DroidBot [29]

As an evaluation metric, we used statement coverage, which has been extensively used to determine the effectiveness of testing tools and is regarded as a good predictor for fault detection [61]. To obtain code coverage, we instrumented the apps using DM-2 tracking statements.

Finally, we used the testing tools to execute each app for 1 hour on a real Google Nexus 5X device, and we repeated the experiments ten times per app to mitigate noise.

3.6.2 Dataset

We evaluated all tools on 12 different apps, randomly chosen from [53], shown in Table 3. More information regarding each app, including its category and number of downloads, is available in Appendix A.

Table 3: Set of benchmark apps to evaluate DM-2

App	**Source**	**Stmts**
Alogblog	F-Droid	428
KeePassDroid (2.0.6.4)	F-Droid	43
BART Runner (2.2.6)	F-Droid	8125
Jamendo (1.0.4)	F-Droid	9347
DroidWeight (1.3.3)	F-Droid	4279
Pizza Cost (1.05-9)	F-Droid	1240
Munch (0.44)	F-Droid	7173
Mirrored (0.2.9)	F-Droid	2475
World Weather (1.2.4)	Play Store	4116
SyncMyPix (0.16)	Play Store	10084
Der Die Das (16.04.2016)	Play Store	3225
wikiHow (2.7.3)	Play Store	3703

3.6.3 Results

Figure 5 shows the average code coverage obtained by the testing tools on our dataset. After 1 hour, DM-2 achieves an average coverage of 49% while DroidBot reaches 41% and Monkey 28%.

DM-2 not only obtains more code coverage than DroidBot and Monkey, but also does it faster. After 1 minute of testing, DM-2 achieves 34% code coverage, DroidBot 26% and Monkey 21%. After 5 minutes, all tools achieve 85% of their maximum coverage—42%, 35%, and 25%, respectively.

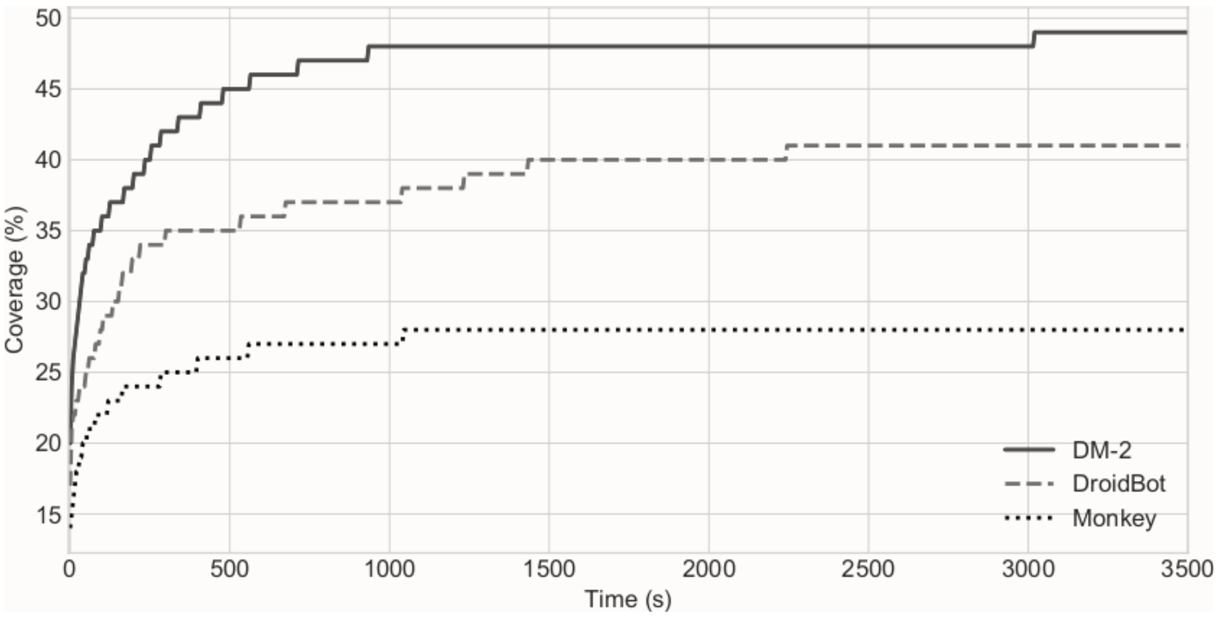


Figure 5: Average coverage over time between DM-2, DroidBot and Monkey for the test dataset.

Considering individual apps, DM-2 achieves more coverage than the other tools on six apps (50%), draws with DroidBot on 1 (SyncMyPix), and with Monkey on 1 (Pizza Cost). DroidBot achieves more coverage in 2 apps (KeePassDroid and DroidWeight), and Monkey achieves more coverage in 2 apps (Jamendo and Der Die Das).

Our results indicate that concrete strategies are superior to purely random events for the tested app set. Even though Monkey implements more input types (e.g., swipe and pinch) than the other tools, it has the worst overall coverage. This is particularly true for apps that have "deeper functionality", requiring the user to navigate through a few screens until it can access certain features.

DroidBot and DM-2, both, try to explore different UI elements systematically. DroidBot applies a depth-first strategy; meanwhile, DM-2 used its *Biased-Random* approach. This fact, together with the better performance of DM-2 (2s instead of 3-4s per action), leads to faster and more efficient explorations.

Out-of-the-box, DM-2 achieves 8% more coverage than DroidBot and 21% more than Monkey after 1 hour. After 1 minute DM-2 already outperforms DroidBot by 8% and Monkey by 13%.

Monkey and DroidBot, however, outperformed DM-2 on two apps each, and we manually inspected what caused this behavior. Both apps where Monkey outperformed the other tools, as exemplified by the coverage from the *Der Die Das* app shown in Figure 6, used the same search bar to trigger different functionality. DM-2 and DroidBot attempt to prioritize behavior that has not yet been covered and, thus, avoided interacting multiple times, which the search bar, leading to worse results.

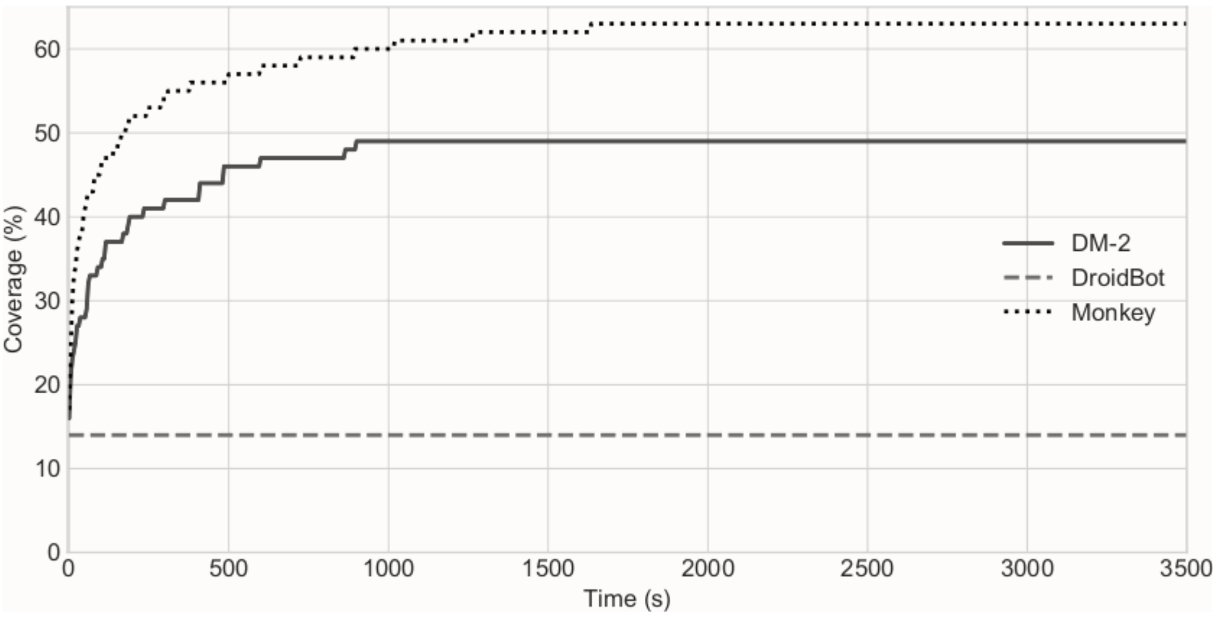


Figure 6: Coverage over time between DM-2, DroidBot and Monkey for app Der Die Das

The apps in which DroidBot outperformed the other tools had context-sensitive behavior that was better addressed by DroidBot's depth-first-search approach. *DroidWeight*, whose exploration is shown in Figure 7, requires the current weight to be defined before calculating the body mass statistics. DM-2's biased-random approach was unable to interact with the app in the correct order.

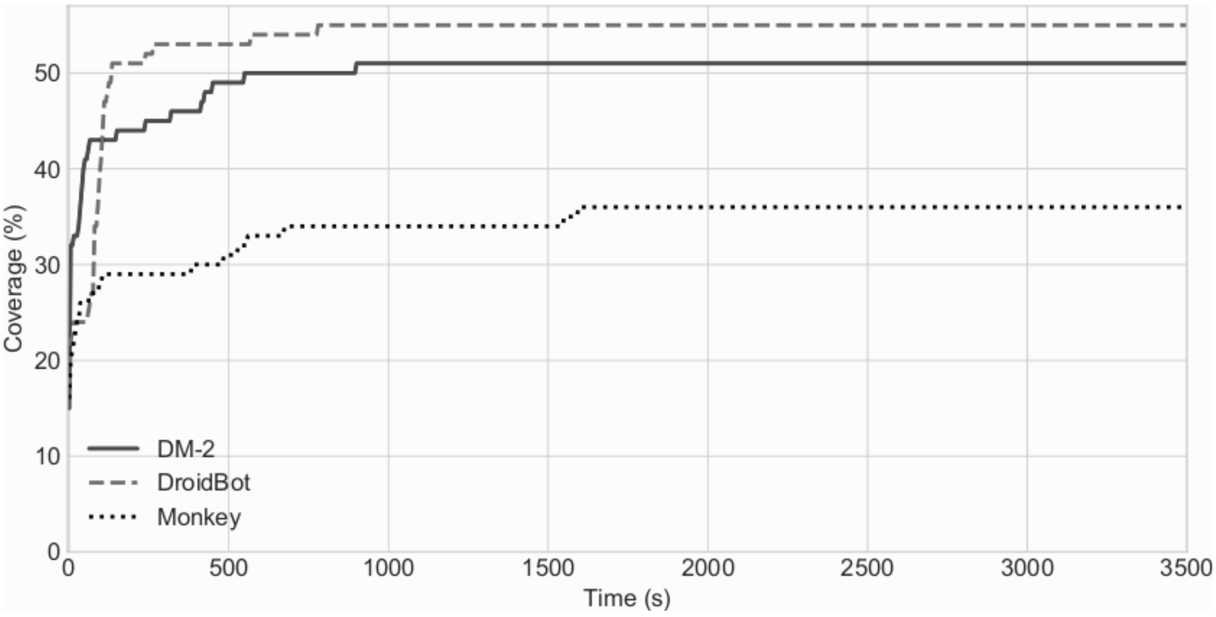


Figure 7: Coverage over time between DM-2, DroidBot and Monkey for app DroidWeight

These outliers highlight one of the main motivations behind DM-2's development. While DM-2's approach worked well on most apps; different strategies better tested some apps. DM-2 allows developers to easily create their testing strategies and combine them with others to test their target apps better.

3.6.4 Threats to Validity

Regarding external validity, we cannot ensure that our results generalize to all apps and testing tools, due to the size of our dataset. To mitigate this threat, our benchmark apps were taken from previous research, which sampled from a variety of sources, commercial (Google Play Store) and open-source (F-Droid); and from a variety of different categories. For further confidence in external validity, more evaluations against other tools and on larger test sets are necessary.

Regarding internal validity, we only instrumented Java byte code to measure the coverage and, therefore, not measured the coverage of other parts of the app code, such as web content and native code. While there is a strong correlation between finding faults and the code coverage of a test suite, the use of a curated repository of bugs could provide more accurate results regarding our techniques.

3.7 Limitations

While DM-2 is a platform that allows the creation of test generators, it cannot test all types of Android apps. DM-2 relies on Android UI Automator, and thus inherits its limitations. UI Automator uses the accessibility API to interact with apps; thus, it cannot interact with all apps. Apps that do not support accessibility, such as games or those which directly draw objects on the screen, cannot be tested. Apps, or widgets, which disable the standard Android accessibility mechanism for security, i.e., banking, are also invisible to UI Automator and, thus, to DM-2. One could mitigate this limitation by combining the inputs of UI Automator with those obtained through image processing techniques [65, 66]. Nevertheless, even images are unavailable for secure screens, such as those from banking, which would remain invisible.

DM-2 also ships out-of-the-box with semi-random exploration strategies. Thus, it is unlikely to reach app functionality that relies on complex inputs, such as a username and password, or human intelligence, such as ordering a product. It can, however, be extended with exploration strategies that handle complex inputs by, for example, exploiting natural language [67]. DM-2's default random-biased strategy will also not work with apps that reuse the same UI element to trigger multiple functionalities. This scenario occurs mostly with apps that reuse the same search bar (input field) and trigger different functionality depending on the content of the input. This situation happened in our experiments and was illustrated in Figure 6.

Concerning performance, DM-2 is limited by both its UI stabilization heuristic and the app speed itself. Therefore, it cannot issue events as fast as tools without UI synchronization, such as Android Monkey. After interacting with the app, DM-2 waits for its UI to stabilize and only then fetches its contents and decides on the next action. If an action relies on external resources, due to a slow Internet connection, for example, DM-2 will wait, while Monkey will create random interactions using coordinates. Our experiments show that DM-2 heuristics outperformed MONKEY when measuring code coverage. Nevertheless, apps whose UI stabilize quickly and where most of the UI is actionable, such as calculators, would be tested faster by MONKEY.

Finally, the code coverage mechanism used by DM-2 relies on Soot [63] and APKTool [62], which work by reverse-engineering the app source code and the compiled Android APK file. As of May 2020, both tools were not fully compatible with the latest format changes from Android 10, and many new app versions could not be instrumented. However, this issue should be solved when these tools are updated to support the latest Android version.

3.8 Related Work

As a test generator, DM-2 is loosely related to the approaches we presented in Section 2.2. The closest work to DM-2 is PUMA [27], a programmable framework whose goal is to abstract the app execution logic from the app analysis one. DM-2's approach and goal, however, significantly differ from PUMA's. We presented DM-2 as a platform for Android test generation as it stands in between being a test generator and being a framework to build new tools. DM-2 follows a framework-oriented design, where it works as a fully functioning template application, and the development of new test tools consist of inheriting and plugging new customized objects. Therefore, differently from PUMA, DM-2 can be used out-of-the-box as a test generator. Moreover, while PUMA required developers to write their analysis tools in its programming language (PUMAScript), creating a DM-2 extension requires developers to implement only their custom behavior by inheriting the respective interface with any Java Virtual Machine compatible language.

Random testing approaches produce random sequences of events to interact with the app. DM-2 is closely related to *random* approaches concerning its out-of-the-box exploration strategy, as it also produces random sequences of inputs. Compared to tools such as DROIDMATE [34], from which it was forked, DYNODROID [31], and MONKEY [30], DM-2 not only offers more functionality—e.g., test reproducibility and code coverage—but also outperforms them due to its more accurate app modeling.

Model-based testing approaches infer models from applications using static or dynamic analysis and use them to generate test cases. *AndroidRipper* [35], A^3E [40], *CuriousDroid* [50], and *Droidbot* [37] produces an on-the-fly dynamic model of the app while testing. *ORBIT* [42] and *SmartDroid* [43] follows a similar approach but uses static analysis to optimize the testing time. DM-2's internal app modeling is inspired by these earlier *model-based* approaches, specially ANDROIDRIPPER and DroidBot [37]. Similarly to these works, DM-2 learns a model of the app during testing and consumes it to generate inputs. However, it employs a heuristic to better re-identify states and widgets during exploration. Moreover, while offering similar features to model-based test generators, DM-2's architecture decouples the exploration strategy from the app model, allowing developers to easily change their exploration strategy to consume the model or even replace the model itself for one tailored to its needs.

Systematic testing approaches interact with the app with specific goals. Approaches such as *EvoDroid* [48] and *Sapienz* [49] combine evolutionary algorithms with random fuzzing to improve test coverage, while *ACTEve* [47] and *IntelliDroid* [46] attempt to trigger specific behaviors through symbolic execution. While out-of-the-box DM-2 does not systematically explore an app, its extensive architecture makes it straightforward to support such approaches, develop new ones, or even use them simultaneously while testing a new app.

Finally, besides the performance benefits as presented in Section 3.6, DM-2 offers additional features like reproducibility (record and replay), extensible architecture, and a more robust app model.

3.9 Lessons Learned

This chapter closes the Part I of this thesis, GUI testing on Android. In this chapter, we presented DM-2, our platform for Android test generation. DM-2 abstracts all interactions between the test and device, allowing the developers to focus on developing their testing strategies. Moreover, it introduces the concepts of *strategy and selector pools*, allowing multiple testing algorithms to be combined and activated according to the current app state. Our experiments showed that out-of-the-box, DM-2 achieves more code coverage than other state-of-the-art tools. Finally, we reuse DM-2 throughout the remainder of this work. We reuse it out-of-the-box as a baseline, and we implement the techniques we propose as DM-2 extensions.
Reproducibility: To facilitate the reproducibility of experiments, the tools and dataset used in the evaluation are available online:

`https://github.com/uds-se/droidmate`

Part II

Learning the Language of Apps

CHAPTER **4**

Learning Actionable Widgets

This chapter is taken, directly or with minor modifications, from our 2018 MobileSoft paper *Guiding App Testing with Mined Interaction Models* [68]. My contribution in this work is as follows: (I) original idea of an mining an interaction model from a crowd of apps and using it for testing; (II) idea and development of the exploration strategy; (III) idea and development of the fitness boost; (IV) partial evaluation.

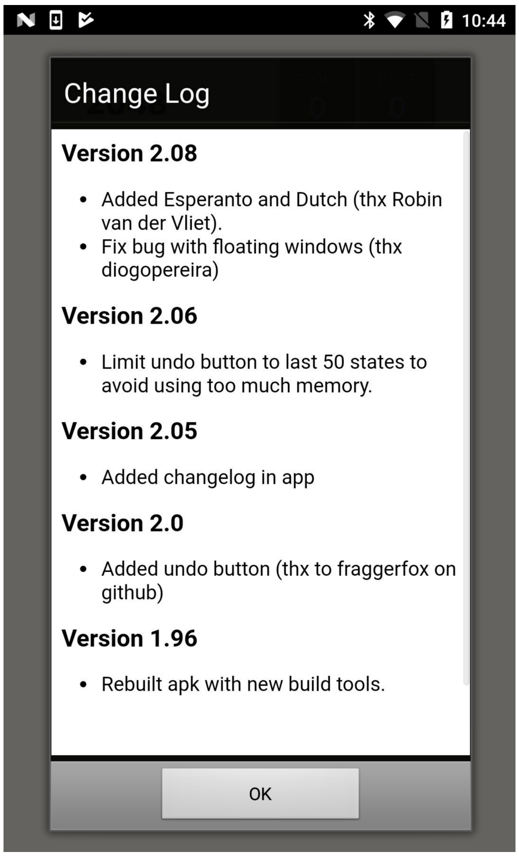
To interact with apps, test generators must synthesize inputs to cover behavior as effectively and efficiently as possible. However, apps are developed to be used from a visual perspective, i.e., a user interacts with an app by visually inspecting its UI. This presents a challenge to test generators as most elements on a user interface are not visible to the user, but instead used to structure how other elements are displayed.

It is typically straightforward for any user with minimal experience of mobile apps to identify which elements on the screen they can interact with. As a simple example, consider the app screen shown in Figure 8a, showing the change log in the popular Android app *2048 Puzzle Game*[11]. On this screen, one may scroll through changelog by swiping up and down, and eventually, dismiss it pressing OK. A user does not even need to know the UI semantics to know which widgets it can interact with. When faced with the same app screen, but in an unknown language (Figure 8b), humans still quickly identify that the app is more likely to react if they interact with the button, instead of the labels.

For a test generator, though, nothing of this is obvious. The individual changes may be defined as clickable; the list is scrollable, but in what direction, the "Change Log" title at the top might be long-clickable, and who knows what happens if one swipes across the OK button. From the standpoint of a test generator, all of these interactions are equally likely to cover some additional app behavior. Consequently, each of the 20 elements shown is equally likely to be clicked, but only one (the OK button) will make progress—95% of the interactions will not lead to progress.

In principle, we may be able to query and analyze the application for active elements. However, user interfaces may be composed at runtime, and not even use standard UI elements: The

[11]https://f-droid.org/packages/com.uberspot.a2048/


10:44

Change Log

Version 2.08

- Added Esperanto and Dutch (thx Robin van der Vliet).
- Fix bug with floating windows (thx diogopereira)

Version 2.06

- Limit undo button to last 50 states to avoid using too much memory.

Version 2.05

- Added changelog in app

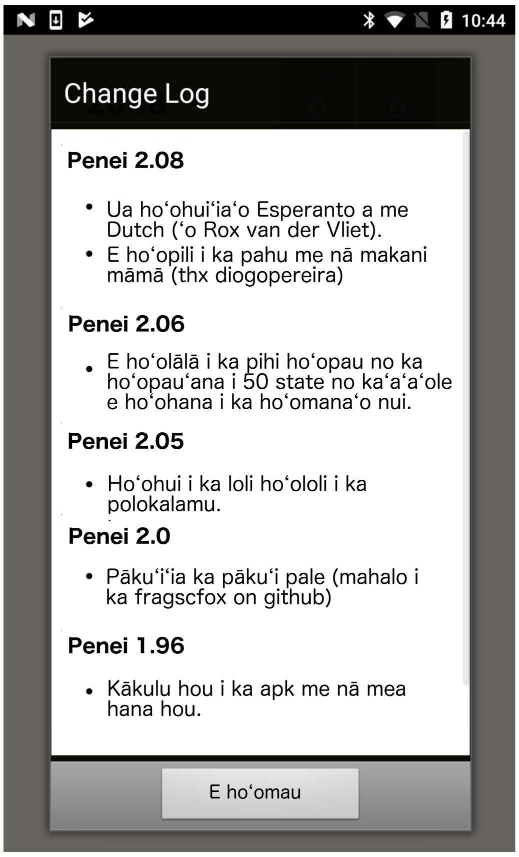
Version 2.0

- Added undo button (thx to fraggerfox on github)

Version 1.96

- Rebuilt apk with new build tools.

(a) Change log screen from the app 2048 Puzzle Game in English


10:44

Change Log

Penei 2.08

- Ua hoʻohuiʻiaʻo Esperanto a me Dutch (ʻo Rox van der Vliet).
- E hoʻopili i ka pahu me nā makani māmā (thx diogopereira)

Penei 2.06

- E hoʻolālā i ka pihi hoʻopau no ka hoʻopauʻana i 50 state no kaʻaʻaʻole e hoʻohana i ka hoʻomanaʻo nui.

Penei 2.05

- Hoʻohui i ka loli hoʻololi i ka polokalamu.

Penei 2.0

- Pākuʻiʻia ka pākuʻi pale (mahalo i ka fragscfox on github)

Penei 1.96

- Kākulu hou i ka apk me nā mea hana hou.

(b) Change log screen from the app 2048 Puzzle Game in Hawaiian

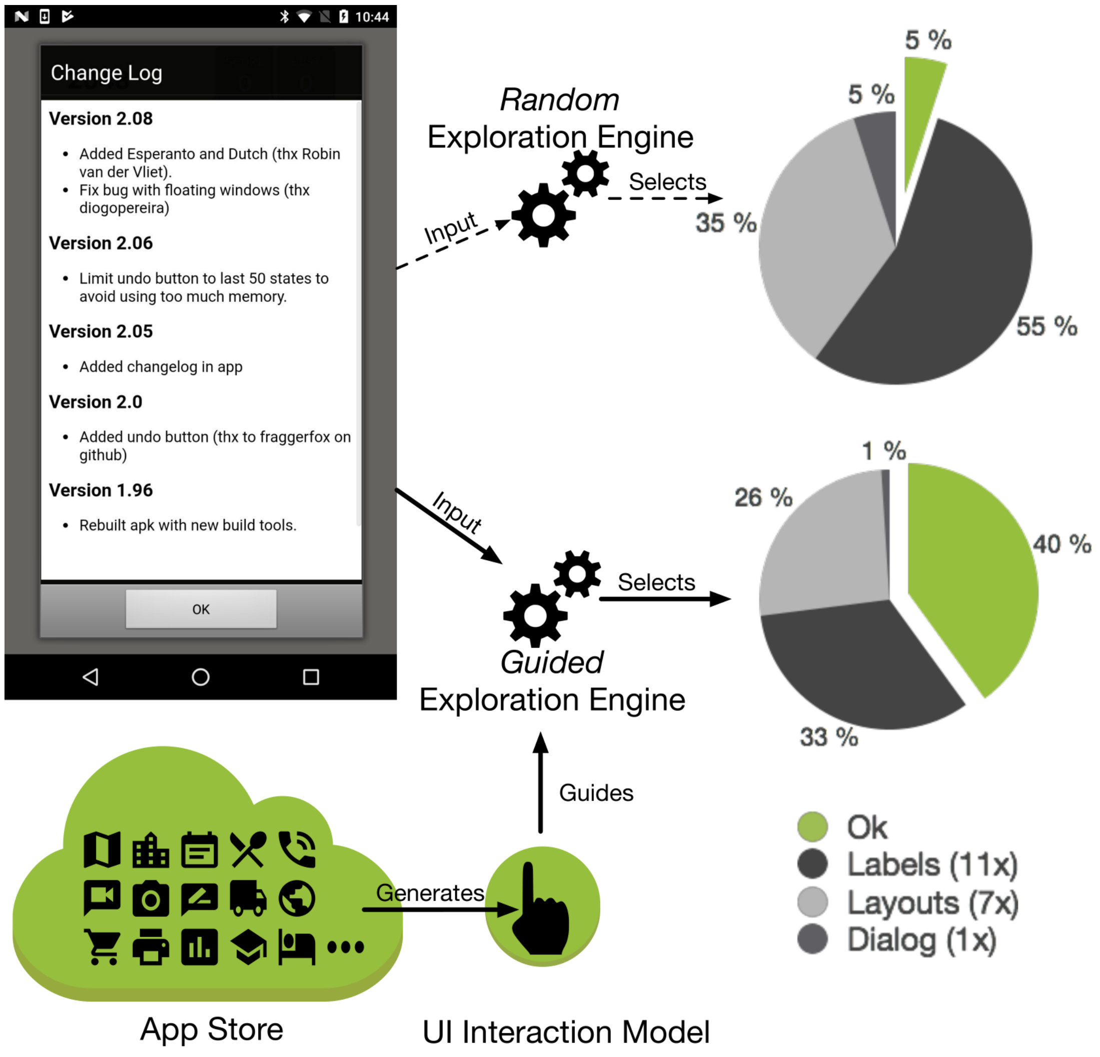
Figure 8: User interfaces from two apps with different interaction possibilities.

2048 change log shown in Figure 9 is actually *web content*, which may tie in JavaScript functions for further interaction both with the Web and the app, posing severe problems for analysis. In all generality, determining whether and how an app responds to an interaction is an instance of the halting problem.

Humans do not require knowledge about an app's inner workings to interact with it. They instead learn over time *conventions* followed by user interfaces. Such conventions may depend exclusively on the widget—buttons are always likely to be clickable—or may depend on where the widget is placed—labels are not clickable unless they are shown inside a list. They may even be imperceptible to the user—clicking on the rectangular area beneath a circular image still works.

In this chapter, we hence follow a simple yet effective approach. We *direct test generation towards UI elements that are most likely to be reactive.* To determine the association between elements and interactions, we statically learn a *UI interaction model* from a crowd of apps. When a UI element is found during app testing, the UI interaction model predicts how likely that UI element is to react to an action (have an event attached). These probabilities can then guide UI exploration towards those UI elements and interactions with the highest probabilities of success.

As an example, reconsider the *2048* screen in Figure 9. Here, we find that the individual changelog entries are *labels*, which are unlikely to respond to clicks, whereas the OK button is a *button*, which typically *does* respond to clicks. Traditional GUI testing (top) randomly distributes generated events across all visible UI elements, leaving only a 5% chance to click on the single active OK button. Our approach (bottom) learns from existing apps which UI elements typically accept interactions (and if so, which ones) into a UI interaction model. Using this model for guidance, UI elements that likely are active (such as OK) are hit much more frequently, resulting in overall faster exploration and testing. Guiding the previously random exploration towards the (likely) button and away from the (unlikely) labels, we can increase the chances of clicking the OK button eightfold, or to 40%. Such guidance can be applied whenever the test generator has to choose from multiple elements; as we show in this chapter, this increases both the effectiveness and efficiency of testing.


Change Log

Version 2.08
- Added Esperanto and Dutch (thx Robin van der Vliet).
- Fix bug with floating windows (thx diogopereira)

Version 2.06
- Limit undo button to last 50 states to avoid using too much memory.

Version 2.05
- Added changelog in app

Version 2.0
- Added undo button (thx to fraggerfox on github)

Version 1.96
- Rebuilt apk with new build tools.

Figure 9: Guiding Test Generation with Mined Interaction Models

We organize the remainder of this chapter according to our contributions. We first introduce how to mine a *UI interaction model*, which captures the normal behavior of UIs in Android apps (Section 4.1). We then show, in Section 4.2, how to consume such knowledge to *guide test generation* towards UI elements and interactions most likely to trigger app behavior. To demonstrate how the *UI interaction model* is abstract and can be attached to arbitrary test generation approaches, we extend two state-of-the-art testing tools: DM-2 and DroidBot.

In Section 4.3, we present the experiments we performed to evaluate how efficient and effective it is to leverage mined interaction models, with results showing that our crowd-based approach improves the coverage of explorations by an average of 20%. After discussing limitations (Section 4.4) and related work (Section 4.5), Section 4.6 concludes the chapter.

4.1 Mining UI interaction models

Traditional model-based approaches (e.g., [36, 43]) extract a specific model for each app to test. Humans, however, do not learn how to interact with a new app from scratch. They instead reuse knowledge from other apps they previously used.

We propose to use a similar approach to test apps. Instead of analyzing the app under test or blindly interacting with its UI, we propose the use of a *universal interaction model* to predict how likely each element on a UI is of reacting to an interaction. We build such a *UI interaction model* using *machine learning* to automatically learn common UI behavior patterns from a crowd of apps. The strength of this approach is that the model is *only learned once* and can be reused with different apps and testing tools, even if the app itself does not support analysis.

We aim to identify different types of UI elements (*e.g.*, *Button*, *TextField*, *Layout*, etc.) and if they respond to any type of event (e.g., *Click*, *Long-Click*, *Scroll*, etc.). Our goal is to extract sufficient information to represent how developers implement interactions with users by using a crowd of apps. Our approach to generating the *UI interaction model* takes four steps, as illustrated in Figure 10, each of which we detail in the remainder of this section.

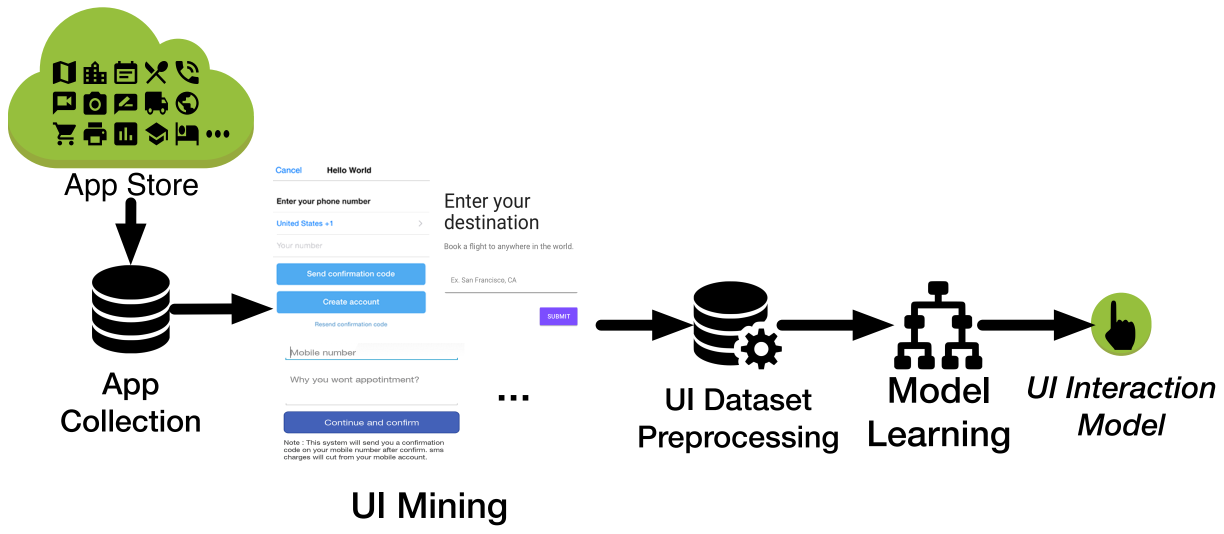


Figure 10: Model generation overview

4.1.1 App Collection

We start by collecting apps available in app stores. In particular, we downloaded 200 apps from the Google Play Store, the official market for Android apps. We opted for apps in the Google Play Store due to their widespread usage among users. While a random selection of apps is necessary to train a statistically representative model of app behavior throughout the whole app store, we opted to create a model that represents widely used apps.

Many apps in Play Store are small apps with few downloads available in a minimal amount of devices, while a few, such as Facebook, YouTube, and Instagram, are installed on over 50% of the devices [69]. Therefore, we randomly selected apps among the top 10 in each of the 32 categories, excluding games, as in [34][12].

We rely on the existence of design guidelines [70, 71, 72] and on the notion that apps tend to follow standard design practices [73] to ensure that a model trained from such apps effectively generalizes to other apps on the Play Store.

4.1.2 UI Mining

After collecting the 200 apps, our next step is to statically analyze each app to extract their widgets and associated code. Since we selected commercial apps, we cannot directly access their source code to determine which event handlers are associated with each widget. We, therefore, analyzed the apps' binaries (*apk files*) using the static analysis tool BACKSTAGE [74]. For each widget, we extracted the following features: *UI type*, *parent element*, *children elements*, *text*, and *event handler*.

Note that some of the extracted features may not have a value (parent or children). Additionally, static analysis cannot guarantee that all widgets and event handlers in the app are mapped. For example, some widgets are created with dynamically loaded content. Our approach, however, does not rely on precisely identifying all possible widgets and their associated event handlers, needing instead only sufficient elements to enable the machine learning algorithm to identify patterns.

From the 200 apps, Backstage identified 119,397 unique widgets and 6 different types of event listeners on our dataset, namely: `onClick`, `onLong Click`, `onTextChanged`, `onScroll`, `onItemTouch` and `onKeyPress`. It associated 9% of the widgets in our dataset with one of these events, that is, interacting with such widget triggers some app behavior. This again highlights the challenge faced by dynamic exploration testing tools that randomly interact with UI elements and the benefits of using a *UI interaction model*. Test generators perform many useless actions, wasting computation time and decreasing their efficacy because many widgets remain unexplored.

Finally, BACKSTAGE mapped 93% of the widgets to *onClick* events, as shown in Figure 11. We, therefore, cannot obtain enough data statically to accurately predict which type of event handler is associated with each widget. Therefore we convert our *event handler* feature from the event type into a boolean indicating if the widget is associated with an event or not.

4.1.3 UI Data Processing

Before learning a model from the mined UI data, we need to preprocess the raw UI information extracted by BACKSTAGE. We perform two tasks:

UI Type Refinement: In Android, developers can specialize the standard UI elements (*e.g.*, Button, TextView) into subclasses. For example, Facebook provides a custom *TextView* named *FbTextView*. Initially, the dataset contained 2,837 different types of UI elements. However, many of them are specific to a small set of apps from the same company.

[12]Google Play Store lists 32 non-game categories at the time of writing.

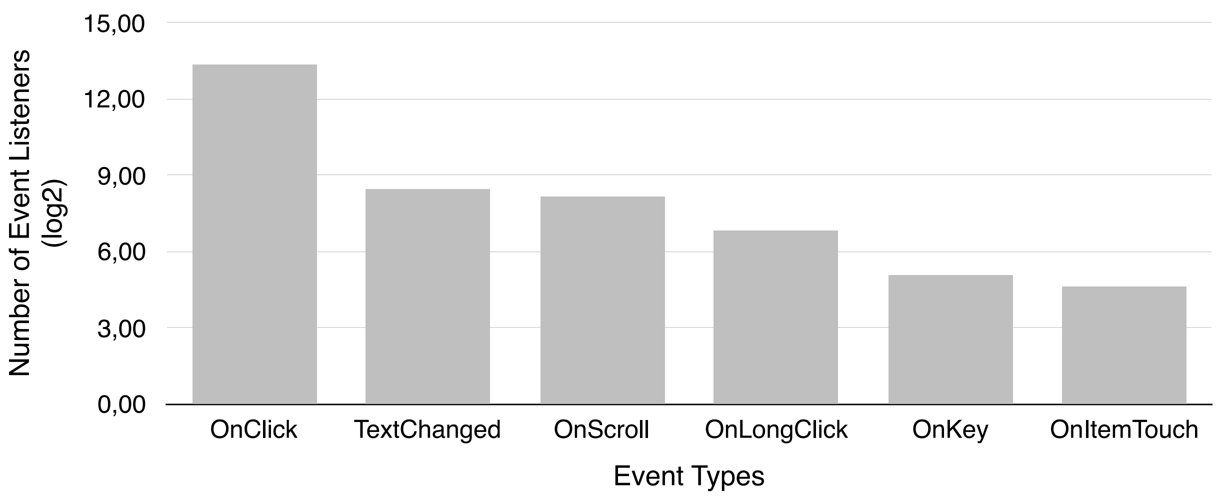


Figure 11: Distribution of event listeners in our dataset

In this task, we map custom widget types to the standard types provided by Android. We first compile a list of native types from the dataset using the package name as a filter. Widgets belonging to Android-related packages (`com.android.*`, `android.*`), are considered as Android native.

We then map the custom widget types to native ones by computing string similarity distance (*i.e.,* Levenshtein Distance) [75]. We do not evaluate if the custom components are subclasses of the native Android ones because some companies opt to inherit their controls directly from the root *View* class, instead of from a specialized Android version. In such situations, however, the name of the components is still similar. The *FbTextView* type, for example, is correctly refined to *TextView* using the string similarity metric, with a class inheritance analysis it would inherit from *View*. After refinement, the dataset contains 108 distinct types of UI elements. Figure 12 shows the distribution of the top 10 most frequent widgets in our dataset. We can observe that *LinearLayout*, mostly structural, and *TextView*, mostly passive, are the most used widgets across apps.

Event Propagation: For each UI element in the dataset that lacks an attached event, we assign the event of its parent widget if it exists. This propagation emulates the Android OS event handling mechanism. When a UI element receives an event, the OS passes the event to the child elements for processing. To facilitate user interaction, developers sometimes associate events with structuring elements (such as *layouts*), instead of with the children elements (such as *images* or *labels*). After this step, the dataset contains ~11% of widgets with an attached event.

4.1.4 Model Learning

Our last step to mine a *UI interaction model* is to train a probabilistic model using the dataset apps. We used the existence of an *event* as the target feature to train our classifier and used the remaining features as independent variables to learn.

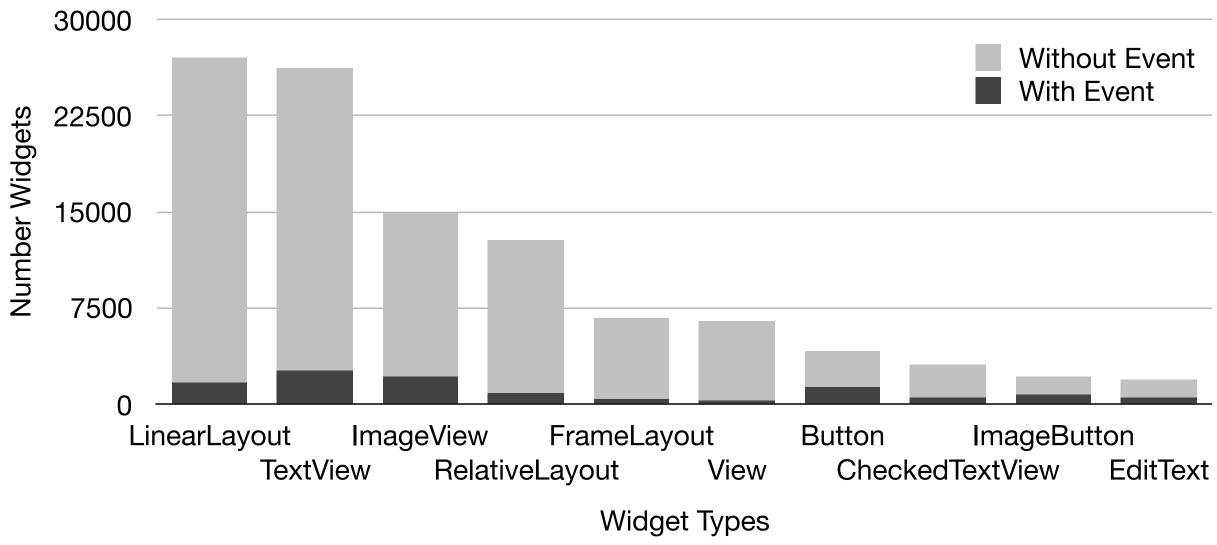


Figure 12: Distribution of widgets with events (top 10 widget types)

Our dataset is inherently unbalanced, i.e., the majority of widgets lack events. This is due to incompleteness of the static analysis and because the apps create a large number of widgets dynamically. The dataset contains a 1:9 ratio of widgets with and without event, as previously shown in Figure 12. For example, 93.65% of *LinearLayouts* lack events attached. On the contrary, 32.53% of *Buttons* have an event. Intuitively, almost all buttons should have an event handler attached. However, layouts can be dynamically composed by other layouts in Android UIs. Dynamically composed UIs cannot be resolved statically without over-approximation. This is the rationale for having a reduced number of buttons with events.

To mitigate the effects of having an unbalanced dataset and avoid over-fitting, we select a *spread subsample* of the data to train the model. In particular, we train the model with the same number of instances with and without an event (1:1 ratio), that is, we select all 10,938 widgets with an associated event, and a random subset of 10,938 widgets without an associated event.

We then used a *Random Forest* [76] classifier to produce a model with determines if widgets are associated or not with an event handler (*UI interaction model*). A *Random Forest* classifier consists of multiple independent tree-based learners, which classify an instance by bagging the results of the individual classifiers. We opted for the *Random Forest* approach for our classifier because it is fast, provides good accuracy results in general, and can deal with unbalanced and missing data. In a preliminary exploratory phase, we experimented with different machine learning classifiers, including SVMs, rule-based, and tree-based models [77], but a *Random Forest* outperformed them.

4.2 Guiding Test Generation with UI Interaction Models

The *UI interaction model* predicts if a widget is likely associated with an event listener or not. In this section, we explain how to use this information to guide the input generation towards those widgets with higher chances of reacting to UI interactions.

Our *UI interaction model* could be used to classify all widgets on a screen as True (has event) and False (no event), and one could filter only the widgets likely to have an event as candidates for interaction. By doing so, however, the test generator would be restricted by the model, instead of guided, and some actions may never be performed. Such a scenario can arise because the model is built on top of approximated data (static analysis) or because some apps may have slightly different ways of interacting. We instead use, for each widget, the *class membership probability* for the True (has event) class as fitness. We do not use the classification result per se but use how confident the classifier is that the widget belongs to the class.

In the introductory example of the Change Log screen in the *2048* app (Figure 9), the UI contains 20 UI elements (1 *DialogBox*, 7 *Layouts*, 11 *Labels*, and 1 *Button*). However, only one element, the *OK button*, is attached to an event listener. The remaining 19 elements structure or display information. With a random input generation approach, each UI element has a 5% selection probability, which results in a 95% chance of selecting a useless element.

According to the model, the *Dialog* has a probability of 2.2% to be associated with an event, whereas the OK *Button* has a probability of 90%, as illustrated in Table 4.

Table 4: Fitness for UI elements at 2048's Change Log screen

	Dialog 1	Layout 1	... ...	Layout 7	Label 1	... ...	Label 11	Button OK
Fitness	0.022	0.063		0.083	0.027		0.068	0.900
Sum	2.250							

4.2.1 Fitness Proportionate Selection

We use a *fitness proportionate selection* approach from genetic algorithms to select UI elements considering their fitness values and thus to bias explorations in favor of relevant UI elements.

The *fitness proportionate selection* algorithm randomly selects an element (p_i) with probability proportional to its fitness (f_i) in relation to the overall fitness of the population. That is, $p_i = f_i / \sum_{j=1}^{N} f_j$, where N is the number of actionable UI elements in the screen under analysis. In the *2048* app example, the *OK* button has a selection probability of 40%: $p_{OK} = 0.900 / 2.250 = 40\%$.

This method simulates a roulette wheel with sectors of size proportional to p_i. Selecting an element is equivalent to choosing a point randomly on the wheel and locating its sector.

We execute the traditional version of the fitness selection algorithm n times and *pick the most frequently selected individual.* This allows us to configure how likely widgets with low-fitness are expected to be selected. By selecting a lower value of n, we increase the probability of choosing a low-fitness widget, and by selecting a larger n, the widgets with higher fitness are more likely to be selected[13]. We experimentally defined $n = 10$.

The pie charts in the introduction of this chapter (Figure 9) illustrate the difference between the *random* and the *fitness-biased random* selection mechanisms when exploring the *Change Log* screen. We can observe that the selection probabilities change between the *random* and *fitness-*

[13]The selection probability of the fitness-proportionate selection is not kept for $n > 1$. Instead, the probability is calculated by a multinomial distribution, as demonstrated by [78].

biased random approaches. In the biased approach, the *OK* button possesses a 40% selection probability, despite representing only 5% of the screen elements.

4.2.2 Fitness Boost

The model's predictive power is not perfect (i.e., precision and recall are inferior to 1.0). Moreover, developers may associate event handlers to widgets in ways not captured by the model. To maximize the coverage of tested functionality, we boost the fitness value of unexplored UI elements.

The idea behind this boost is to give these widgets a higher chance of being explored at least once. Before performing each action, the fitness value of each unexplored element is increased by 100%. Once the widget is explored, it no longer receives the boost and retains its original fitness value. Following this approach, the exploration initially prioritizes the "best" elements (according to the model). Then, once it explored these widgets, it increases the chance of interacting with "worse" elements once. Once all elements have been explored at least once, the fitness boost no longer applies, and only the model values are used.

Consider the scenario presented in Figure 9. Assume that the *UI interaction model* returns the following fitness values for the elements on the screen: button = 4, each label = 0.3, each layout = 0.37, and dialog = 0.1. The fitness boost then works as follows:

1. The scenario starts with all widgets receiving the fitness boost because they have not yet been interacted with. There is a 40% probability of interacting with the *OK button*, a 33% probability of interacting with a label, a 26% probability of interacting with a layout and 1% chance of interacting with the dialog.

2. If, as the first action, we interact with the *OK button*, it will no longer be boosted. For the next action, the fitness of the widgets change to button = 4, each label = 0.6, each layout = 0.74, and dialog = 0.2, since all elements but the button are boosted. This results in 41% chance of interacting with a label, 32% chance of interacting with a layout, 1% chance of interacting with the dialog and only a 25% chance of interacting with the button.

3. If we click on the button again, the fitness and probabilities do not change, as the button already did not receive a boost.

4. If, as the next action, we interact with a *label*, only that label will not be boosted, again changing the fitness of the widgets on the screen. For the next action, the fitness of the widgets change to button = 4, clicked label = 0.3, each other label = 0.6, each layout = 0.74, and dialog = 0.2. This results in a 40% chance of interacting with a label, a 33% chance of interacting with a layout, a 1% chance of interacting with the dialog, and only a 26% chance of interacting with the button. Slightly increasing the probabilities of interacting with a layout or again with the button, while decreasing the probability of interacting with a label.

The probabilities of interacting with the elements on the screen are, thus, adjusted after each action, until all elements on the screen have been interacted with. At this point, the fitness boost no longer plays any role in the exploration, and the original probabilities of the model apply.

4.3 Evaluation

The *UI interaction model* aims to be a general model of how likely widgets are to react to UI events. We, thus, conducted a set of experiments to measure its effectiveness and benefits. In the remainder of this section, we present our experimental setup, dataset, and answers to the following research questions:

RQ1 (Model Representativity and Generalization). Is the *UI interaction model* representative of normal app behavior? (Section 4.3.3)

RQ2 (Model Accuracy In Practice). Does the *UI interaction model* (learned statically) effectively predict events dynamically? (Section 4.3.4)

RQ3 (Effectiveness of the Approach). Is the *crowd-based dynamic exploration* more effective than the random and model-based explorations? (Section 4.3.5)

4.3.1 Experimental Setup

To demonstrate that the model can be used alongside different testing tools, we extended both DM-2 [34] and DroidBot [37] with our mined *UI interaction model.* DM-2 implements a pseudo-random GUI exploration strategy, while DroidBot follows a model-based exploration strategy. We chose these tools because their source code is publicly available online, and they can test apps without having access to the apps' source code. The tools represent two well-adopted approaches (*i.e.,* random and model-based) to test Android apps. Along this section, we refer to our extended versions of the tools as Droidmate-M and Droidbot-M.

We executed our experiments on four types of devices: Nexus 5X, Nexus 9, Nexus 6 and Pixel C, all running Android 7.1 (API 25). To prevent inconsistencies coming from different device behaviors, all tests of the same app are performed in the same device model.

4.3.2 Benchmark Apps

We selected 17 apps from [29]for our tests according to the limitations of the test generators used in the experiments. For a varied and representative sample, our selection includes apps from *Google Play Store* and *F-droid*[14]. We selected apps that span over different categories and have different sizes. The use of open-source apps allows us to identify limitations in static analysis. In contrast, the use of commercial apps aims to illustrate that our approach improves tests even when no source code is available.

Table 5 summarizes the set of benchmark apps. It provides for each app its source, number of statements, widgets, and events. We used through DM-2 code coverage instrumentation and the Backstage tool to obtain this information. More information regarding each app, including its category and number of downloads, is available in Appendix B.

[14]F-Droid is an open-source repository of Android apps. https://f-droid.org
[15]These apps crashed when being evaluated with Backstage.

Table 5: Set of benchmark apps

App	Source	#Stmts	Widgets	Events
Alogblog[15]	F-Droid	428	-	-
KeePassDroid (2.0.6.4)	F-Droid	43	169	0
Munch (0.44)	F-Droid	8084	387	0
BART Runner (2.2.6)	F-Droid	8125	170	5
Jamendo (1.0.4)[15]	F-Droid	9347	-	-
2048 (2.06)	F-Droid	168	3	1
DroidWeight (1.3.3)	F-Droid	4279	63	22
Pizza Cost (1.05-9)[15]	F-Droid	1240	-	-
Mirrored (0.2.9)	F-Droid	2475	29	0
Easy xkcd (5.3.9)	F-Droid	13768	265	6
Dialer2 (2.90)	F-Droid	2005	55	19
PasswordMaker (1.1.11)	F-Droid	4378	177	30
Tomdroid (0.4.1)	F-Droid	2727	21	0
World Weather (1.2.4)	Play Store	4116	205	0
SyncMyPix (0.16)	Play Store	10084	81	15
Der Die Das (16.04.2016)	Play Store	3225	69	0
wikiHow (2.7.3)	Play Store	3703	183	7

4.3.3 RQ1: Model Representativity and Generalization

The *UI interaction model* predicts if widgets have an event handler attached. The higher the accuracy of the model, the more effective the succeeding dynamic exploration. In this experiment, we study the prediction accuracy of the inferred *UI interaction model*. We aim to discover if the model is representative of the “normal” behavior of apps.

For this purpose, we assess the model using *10-fold cross-validation* on the training dataset (cf. Section 4.1), and we consider the values returned by Backstage as ground truth. Note that we do not use the actual classification results for test generation, relying instead on the class membership probability. Both concepts are, however, related. This self-test allows us to verify whether and how much the model is representative of app behavior.

Figure 13 shows the performance of the *UI interaction model* using 10-fold cross-validation with the training set. The confusion matrix quantifies the number of correctly and incorrectly classified instances for each class. The overall *precision* (the percentage of classes predicted to have event handlers that do have event handlers) is 68%. In contrast, the overall *recall* (the percentage of classes with event handlers that are correctly predicted as such) is 75%. The precision and recall for predicting the True class (widgets having an event) are 68% and 75%, respectively. Whereas for predicting the False class (absence of event), precision and recall are 72% and 65%, respectively.

	Classified as			
Input	**True**	**False**	**Total**	
True	TP = 8212	FN = 2726	10938	*Precision* = 68%
False	FP = 3858	TN = 7080	10938	*Recall* = 75%
Total	12070	9806	21876	*Accuracy* = 70%
				Specificity = 65%

Figure 13: Confusion matrix for presence (True) and absence (False) of event handlers using 10-fold cross-validation with the training set.

Alongside this experiment, we evaluated how the model would behave using the whole original dataset, without performing the spread sub-sampling to create a balanced dataset (cf. Section 4.1). In this case, the model presented both precision and recall superior to 90%. However, this is due to the 9:1 ratio of widgets without and with an event in the dataset. The resulting classifier classified all widgets as *without event*, resulting in a 0% recall for the False class. Balancing the dataset avoids biased results.

The *UI interaction model*, statically mined from a crowd of apps, has a precision of 68% and a recall of 75% when predicting if unseen widgets are associated with an event handler.

4.3.4 RQ2: Model Accuracy In Practice

The *UI interaction model* is obtained by statically analyzing a crowd of apps. One of the well-known limitations of static analysis tools is that they can infer behaviors that never happen during execution. Besides, the inferred knowledge can be incomplete, as not all behaviors can be statically determined. We aim to investigate if the statically mined *UI interaction model* represents dynamic app behavior.

For this purpose, we evaluate the model's performance at runtime; that is, we investigate if the behavior captured by the model statically represents the behavior of apps dynamically at runtime.

To measure the prediction success dynamically, we compute the number of *effective actions* in the tests. We consider as *effective actions* the app interactions that produce a reaction in the app. In other words, the actions that interact with widgets which have an event listener attached.

Since the code of some apps is unavailable, apps have several thousands of widgets and methods. Each execution trace contains hundreds of actions; it is infeasible to assess each action manually. For this reason, we use a heuristic to compute the number of *effective actions* automatically. Intuitively, if after an action, there is a reaction in the app, then a change will happen on the app screen. Thus, we consider an action as *effective* if the screenshots before and after the action are different.

We measure the similarity between screenshots using an image processing approach similar to [79]. First, we remove the top part of the images corresponding with the device *'status bar'*, as it contains widgets such as the clock and battery level, which could evolve during the execution, thus producing false positives. We then perform an image subtraction for each pair of consecutive screenshots. If both screenshots are identical, we conclude that the action did not produce any reaction in the app—*i.e.,* the action is *ineffective.* We are aware that this measure is an approximation. For example, an action could activate a process in the background without notifying the user. Nevertheless, we expect this situation to happen rarely. One of the most basic principles of *UI design* is providing feedback to users about the system state [80]. We performed a reduced manual evaluation with the subject apps, and concluded that screenshot similarity is a suitable approach to identifying *effective* and *ineffective* actions.

DM-2 provides the feature to capture screenshots after each action. Therefore, we use only DM-2 to perform this experiment. We run DM-2 and Droidmate-M to test the 17 benchmark apps. We set the tools to execute 500 actions (*i.e.,* events) with each app. To reduce the impact of noise, for each app, we repeat the execution ten times and compute the average percentage of ineffective actions.

The guided exploration *significantly reduces the number of ineffective actions* in 13 out of 17 apps. As shown in Fig.14, the number of ineffective actions significantly varies among apps, since it depends on the app's design. On average, DM-2 performs 29.04% of ineffective actions while Droidmate-M 14.80%. Overall, the guided exploration reduces approximately by half the percentage of ineffective actions. The reduction factor ranges from 8% to 80%, depending on the app.

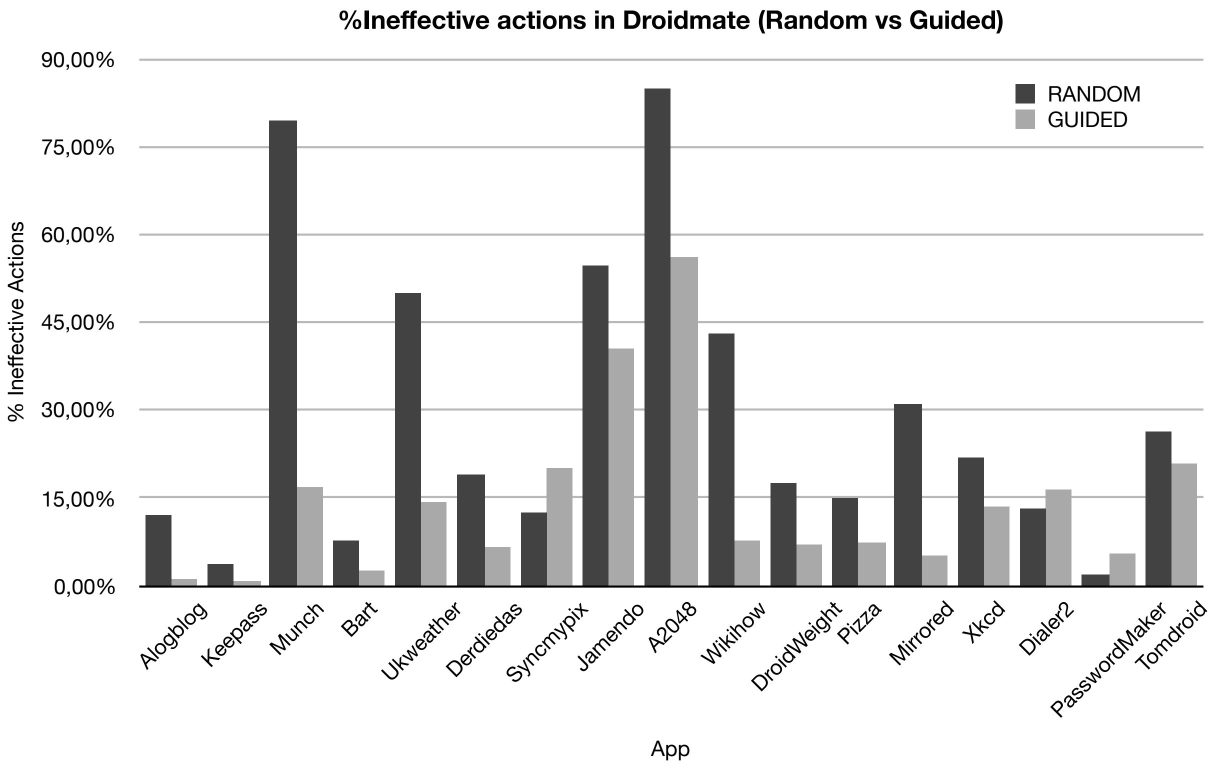


Figure 14: Comparison of *ineffective actions* in DM-2 between Random and Guided exploration

Nevertheless, there are three apps (*Syncmypix*, *Dialer2*, and *PasswordMaker*) where the random exploration executed fewer *ineffective* actions. After manual inspection of the source codes, we observed that in these apps, almost all elements displayed on the screen respond to interactions. Thus, virtually any random interaction is effective. Since the guided exploration gives a boost to unexplored elements, it attempted to interact with all widgets and, thus, executed more ineffective actions.

In conclusion, the effectiveness of our approach will highly depend on the app's design. On the one hand, if all elements in the UI respond to interactions, a random approach will perform well because every input will affect the app. On the other hand, if the UI contains many widgets but few event listeners, then the guided approach will lead to more successful explorations.

On average, explorations guided by our *UI interaction model* reduces the number of ineffective actions by 50%, with reduction factors of 8% up to 80%.

4.3.5 RQ3: Effectiveness of the Approach

Our last experiment aims to quantify the effectiveness of the crowd-based approach compared with state-of-the-art testing approaches concerning how much functionality can be tested.

We compare the performance of the tools DM-2 and DroidBot against their extended versions that use our model—i.e., Droidmate-M and Droidbot-M. Furthermore, we compare them with Monkey [30], the most popular tool to test Android apps. It generates random UI events (*e..g,* clicks, touches) to stress apps, without considering the UI of the apps. Although Monkey implements the most basic strategy, previous research has shown it to outperform other sophisticated techniques [29].

We select code coverage as a metric to evaluate and compare testing strategies, as previous research has demonstrated that it is a good predictor for the test quality [61]. Similar to [29], we compute statement coverage. We use DM-2's native instrumentation mechanism to obtain code coverage without having access to the source code. For DroidBot, we port the code from DM-2's instrumentation into DroidBot to log each statement in the app bytecode.

We run the five tools (DM-2, Droidmate-M, DroidBot, Droidbot-M, and Monkey) with the 17 apps in our benchmark. We execute each app for 25 minutes, where our previous experiments (Section 3.6) showed that all tools already reached over 95% of their coverage. Moreover, to reduce the impact of noise, we executed each app ten times and obtained the average coverage over time.

Guided exploration is more effective than the original random and model-based explorations in DM-2 *and* DroidBot. Figure 15 reports the coverage that each tool achieved with the benchmark apps.

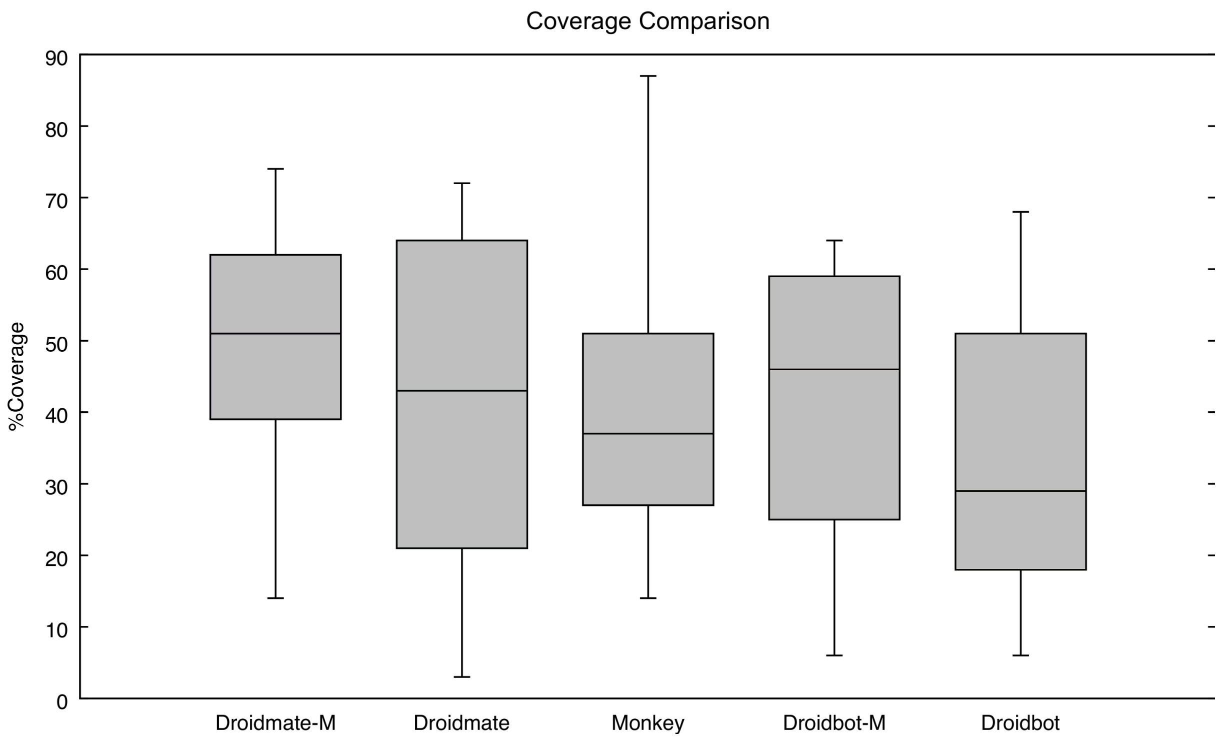


Figure 15: Coverage comparison of *guided* and *random* explorations over the benchmark apps

Note that with DroidBot, the apps *PasswordMaker* and *Tomdroid* crashed during execution, and DM-2 stopped. Thus, the results reported in Figure 15 for DroidBot and Droidbot-M exclude those two apps.

While the median coverage of DM-2 is 43.03%, Droidmate-M reaches 51.38%. Thus, there is a coverage percentage increase of 19.41%. The percentage increase is computed as: $\%increase = (p2 - p1)/p1 * 100$, where $p1$ and $p2$ are the two percentages to compare.

DroidBot obtains a coverage percentage increase of 55.30% when using the guided strategy. From a median coverage of 29.87%, it goes up to 46.39%. This experiment demonstrates that guided exploration improves the performance of random explorations.

Also, Monkey obtained a median coverage of 37.2%. First, we can observe that both DM-2 and Droidmate-M outperform the median coverage of Monkey, reinforcing our previous results (Section 3.6) with a slightly larger dataset. Second, with the larger dataset, Monkey performs better than DroidBot. Nevertheless, when using the guided strategy, Droidbot-M outperforms Monkey and also DM-2.

The state of the art tools DM-2 and DroidBot experience a coverage percentage increase of 19.41% and 43.03% on average when using the guided strategy across all apps.

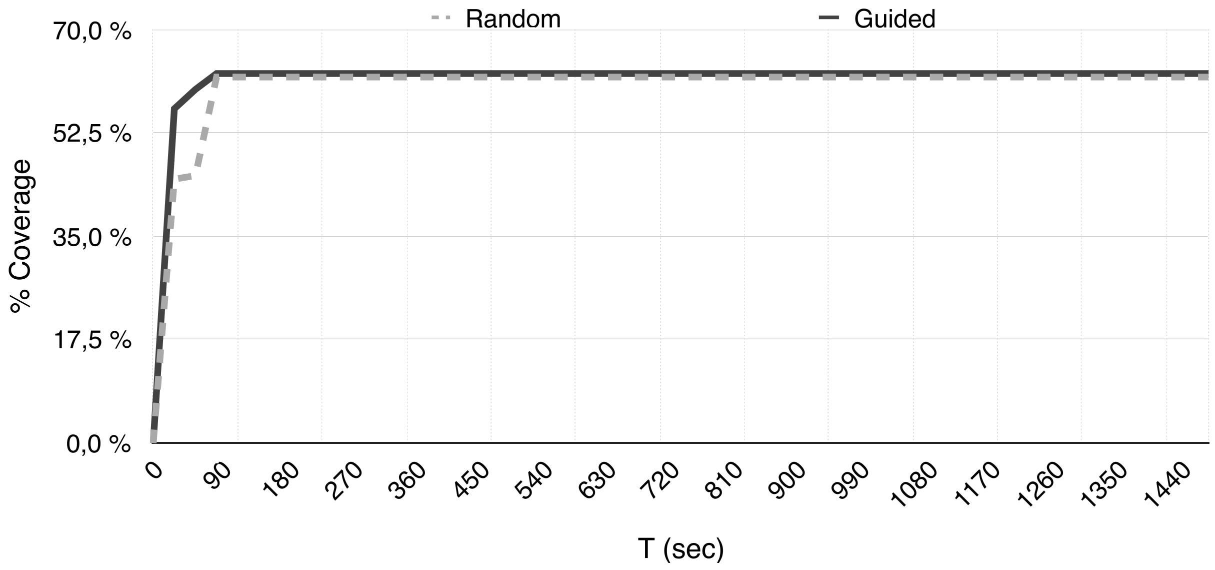


Figure 16: Average coverage over time in DM-2 with *guided* and *random* explorations in the app 2048

We then inspected the individual apps. We first observed four apps (*Aalogblog*, *KeePassDroid*, *2048*, and *Tomdroid*) where all random and guided explorations obtain the same coverage. Since the two different testing methods achieve the same statement coverage, it indicates that no more coverage can be achieved without more advanced testing techniques. We manually checked if, with manual exploration, it is possible to achieve higher coverage. There are login screens in which the testing tools fail to pass, and swipe actions which the testing tools failed to execute. Moreover, for scenarios where both tools reached the same coverage, as exemplified by the coverage evolution of the *2048* app using DM-2 and Droidmate-M in Figure 16, we can observe that the guided exploration reaches its peak coverage faster.

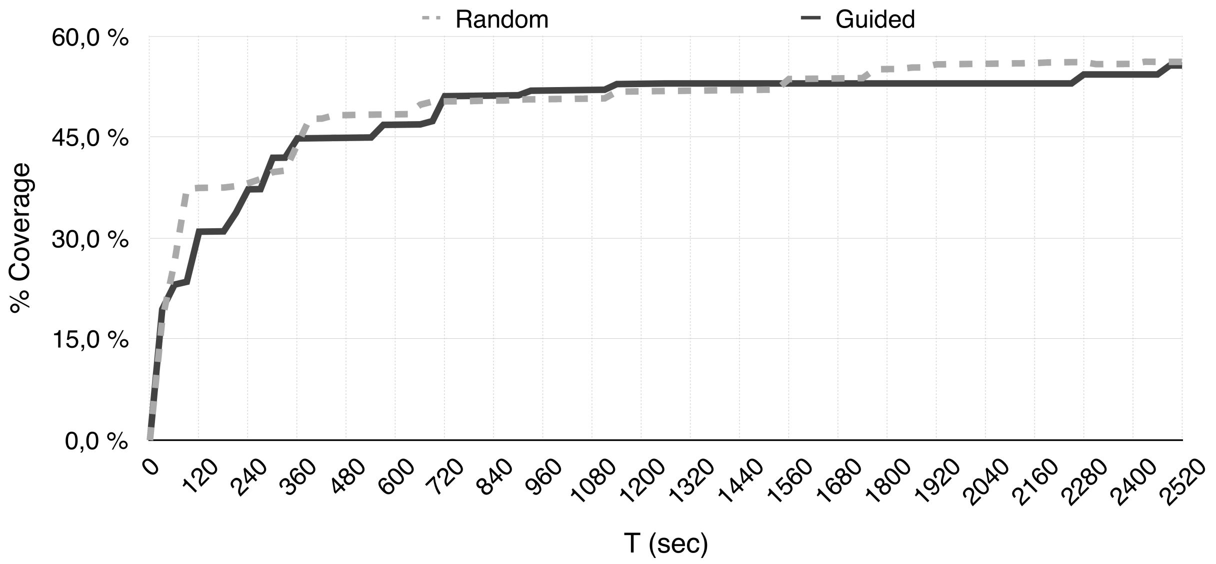


Figure 17: Average coverage over time in DM-2 with *guided* and *random* explorations in the app DroidWeight

We further explored three cases. First, we show a case where the random exploration outperforms the guided exploration and two cases where the guided explorations win. Figure 17 shows the coverage evolution in the *DroidWeight* app, where the random exploration outperforms the guided one. We can observe that random exploration was consistently better throughout the whole exploration. However, while the coverage's curve of random exploration has flattened for approximately 10 minutes, the guided exploration was able to explore new widgets. Finally, both strategies converge.

In the *Jamendo* app (Figure 18), both explorations start with the same coverage, because most widgets in the initial UI possess an event handler. After reaching a screen where some widgets lack an event handler, the guided exploration increased coverage faster than the random one.

As we have previously discussed, the performance of the approach highly depends on the design of the apps. Figure 19 reports the coverage evolution during the exploration of DM-2 and Droidmate-M. The graph shows the average coverage across all benchmark apps over ten runs. We can observe that guided exploration speeds up testing; more code is covered faster. After 10 minutes of exploration, the guided exploration has already obtained a coverage increase of 9.9% compared to the random exploration. This becomes especially relevant in mobile ecosystems, where time is a critical success factor. App updates are frequent [81]; by reducing testing time, release cycles can be accelerated.

The guided exploration speeds-up testing.

Also, our experiments have demonstrated the *applicability* of our model. We have applied the model into two different testing tools (DM-2 and DroidBot), which are implemented with different programming languages (*i.e.,* Kotlin and Python), and follow different testing strategies.

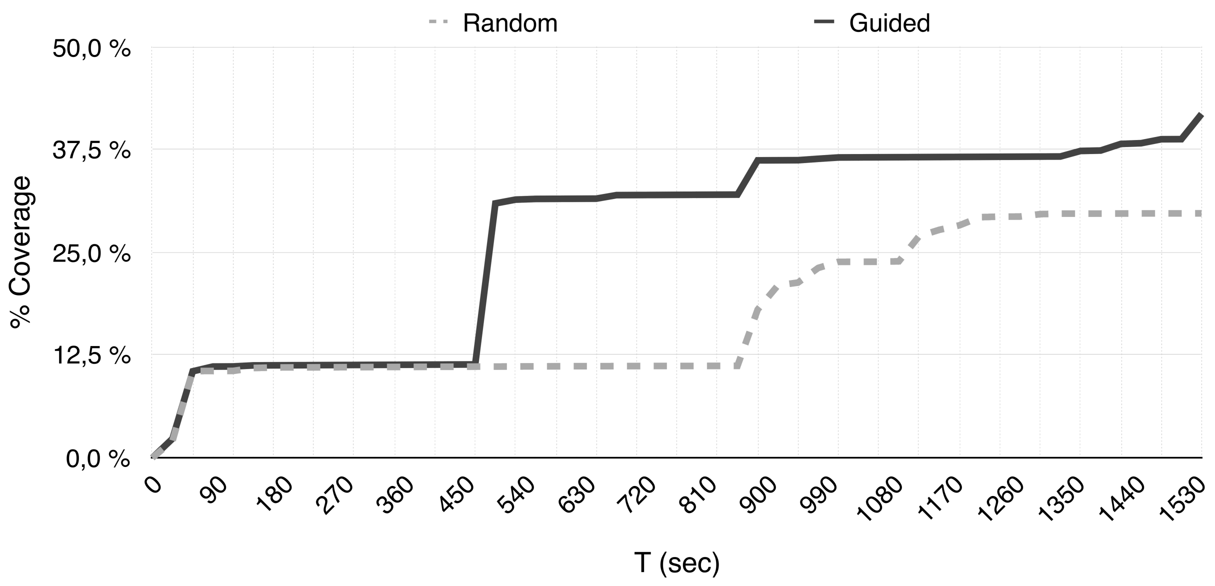


Figure 18: Average coverage over time in DM-2 with *guided* and *random* explorations in the app Jamendo

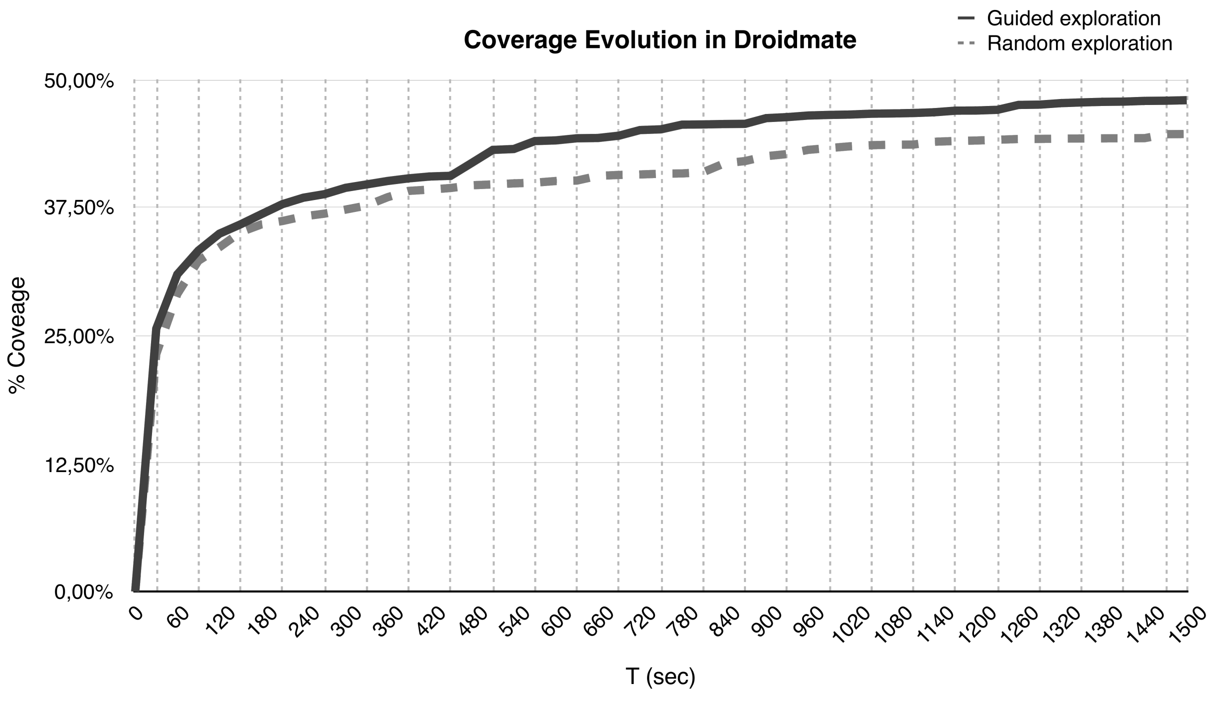


Figure 19: Average coverage evolution in DM-2 with *guided* and *random* explorations over the benchmark apps for 25 minutes

The extensions of these tools to incorporate the guided-strategy comprise ∼100 LOC and took around 3 hours of work in each app.

This increase in effectiveness comes at a price: One must first mine a universal model, as described in this chapter. Even though this model can be reused repeatedly (unless the essential behavior of Android widgets changes over time), one may ask whether this effort is necessary. In particular, one may ask whether it would not be better to mine a *specific* model from one application in order to generate tests for it. Unfortunately, such an app-specific model is not as useful as it may seem, due to the limitations of static analysis. As an example, consider the *DroidWeight* app from our benchmark—an app in which DM-2 (*i.e.,* random exploration) performed better than Droidmate-M, which indicates that the crowd-based model did not accurately represent this app. We statically extracted the widgets and events *from the* DroidWeight *app only* and trained a random forest classifier, similar to the crowd-based model. In other words, we applied our approach *without* leveraging the crowd of apps.

To measure the *effectiveness of the approach*, we executed Droidmate-M with the crowd-based and the single-app models and compared the number of effective actions and achieved coverage. We follow the same procedure described in Section 4.3.4 and 4.3.5; similar to the previous experiments, we execute 500 actions and ten runs.

When using a single-app model, the amount of ineffective actions drops from 7.05% to 3.23%, i.e., a reduction factor of 55%. *The reduction of ineffective actions, however, does not translate into improved coverage.* The crowd-based model achieved a code coverage of 52.93%, while the single-app model achieved only 34.00%. The limitations of the static analysis explain this difference. Widgets that are created dynamically or whose event handler cannot be statically determined, are classified by the model as *absence of event* with almost 100% probability. Thus, those widgets are seldom actioned during exploration. In particular, the crowd-based model executed 211 unique widgets during exploration, while the single-app model actioned 58.

A model generated from a single app can effectively avoid ineffective actions. However, it may be biased due to limitations of static analysis and, thus, achieve less code coverage.

4.3.6 Threats to Validity

Regarding external validity, our experiments have demonstrated evidence that the guided exploration using crowd knowledge significantly improves the testing performance of state of the art tools (DM-2, DroidBot), with a set of benchmark apps. However, we cannot ensure that the results generalize to all apps and testing tools. To mitigate this threat, we selected a varied sample of apps from different sources, including commercial and open-source apps, with different sizes and categories. We applied the approach to two testing tools that implement different exploration strategies and use different programming languages.

The proposed approach is implemented on the Android platform because it is currently the most widespread mobile OS (having +88% of market share [2]). Nevertheless, our conceptual foundations could be applied to other mobile platforms (e.g., iOS) and domains (such as web applications). To gain confidence in external validity, more evaluations are needed on other platforms and tools.

Regarding construct validity, we use a static analysis tool, Backstage, to extract UI information from apps. This information is then used to learn the UI interaction model. The extracted information is incomplete due to the inherent limitations of static analysis (illustrated in Section 4.3) and to the tool's limitations. For example, there is a growing tendency in using third-party annotation libraries to implement event handlers. The annotation libraries aim to simplify and speed-up development. Currently, Backstage only considers the standard event declaration mechanisms provided by Android. The more complete the model, the more effective will be the resulting dynamic explorations. Learning a model from a crowd of apps, instead of a single app, mitigates this threat. The model can be completed with information coming from several apps.

Regarding internal validity, in order to evaluate our approach with both commercial and open-source apps, we performed bytecode instrumentation. Moreover, we measured only the coverage of Java code, which belongs to the application. More precise measurements can be obtained using source code instead of bytecode for instrumentation and by measuring coverage on Javascript, dynamic, and native code.

4.4 Limitations

Our experiments showed the benefits of learning and consuming an interaction model to test apps. Nevertheless, our implementation and approach would not work for all apps. Our implementation inherits all limitations from DM-2, which we previously discussed in Section 3.7.

Additionally, our model assumes that apps have similar interaction patterns. We showed in RQ3 4.3.5 that this assumption does not hold for all apps. If an app does not follow the same patterns as those used during training, the interaction model may have an adverse effect, resulting in a more inefficient test.

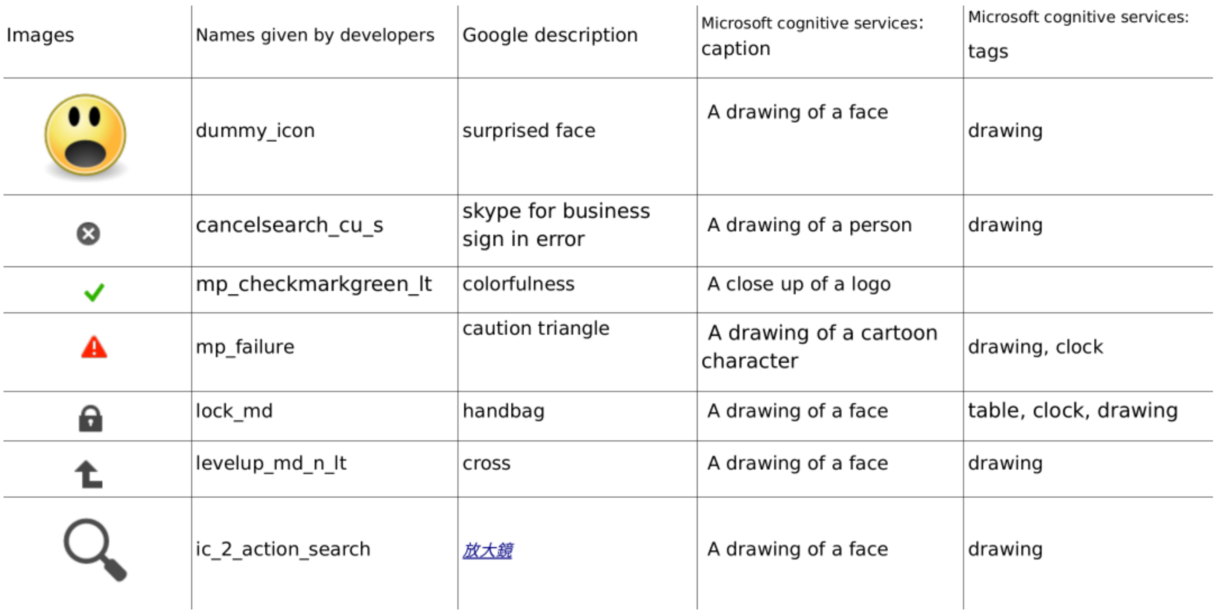
Our interaction model also does not model UI semantics. A human not only relies on structural UI patterns to interact with the app but also on the semantic of its UI elements. For example, it distinguishes between an image that denotes an action, such as a trash bin for delete or a magnifier for search, from a decorative one, such as a logo, even if both follow the same structural pattern. We attempted to remove this limitation by using commercial image semantic extractions tools. Namely: Microsoft Cognitive Services[16] and Google Reverse Image Search through its Custom Search API[17]. These tools, however, failed to provide adequate semantics, as shown in Figure 20. How to extract semantics from arbitrary images is still an open problem.

4.5 Related Work

This section summarizes the most relevant works related to this research.

In Section 2.2, we presented existing test generation approaches, which are loosely related to our work. Closest to the approach we presented in this chapter is iMPAcT [39], which identifies the UI patterns used in each app screen and compare them to an internal catalog of patterns to determine how to interact with the app. Similar to iMPAcT, we also learn how to interact with an app based on a catalog of patterns. However, we do not determine which patterns to implement and how to interact with them. Instead, we rely on patterns learned by a classifier trained on a dataset mined from a crowd of apps. Nevertheless, in general, most test generators can benefit from knowledge about how to interact with different UI elements. Such information can be used to prioritize UI elements that are more likely to trigger app behavior or to guide the exploration towards the desired path.

[16]`https://azure.microsoft.com/en-us/services/cognitive-services`.
[17]`https://developers.google.com/custom-search/v1/overview`.



Images	Names given by developers	Google description	Microsoft cognitive services: caption	Microsoft cognitive services: tags
😮	dummy_icon	surprised face	A drawing of a face	drawing
⊗	cancelsearch_cu_s	skype for business sign in error	A drawing of a person	drawing
✓	mp_checkmarkgreen_lt	colorfulness	A close up of a logo	
⚠	mp_failure	caution triangle	A drawing of a cartoon character	drawing, clock
🔒	lock_md	handbag	A drawing of a face	table, clock, drawing
⮤	levelup_md_n_lt	cross	A drawing of a face	drawing
🔍	ic_2_action_search	放大鏡	A drawing of a face	drawing

Figure 20: Attempt to retrieve image semantics from commercial tools

Random testing approaches, such as Monkey [30], Dynodroid [31], and DM-2 [34] create random sequences chains of events to explore an app. Our approach differs from the traditional random test generators by giving preference to UI elements that have a higher probability of reacting to an event. It can be used alongside semi-random algorithms, such as Dynodroid's prioritization of least-explored widgets. Nevertheless, in a scenario where all UI elements have the same probability of containing an event, the approach behaves equally to a purely random.

Model-based testing tools infer models from applications using static, dynamic, or a hybrid analysis and use them to generate test cases. GUIRipper [35] and MOBIGuitar [36] dynamically traverse an app's GUI and create a state machine model that it later uses to generate test inputs. ORBIT [42] follows an approach similar to GUIRipper and MOBIGuittar but uses static analysis to reduce the number of GUI elements to test. SmartDroid [43] also exploits static analysis to generate activities and function call graphs to identify paths that should be explored. SwiftHand [38] uses machine learning to create a model of the app and consumes this model to generate inputs to visit unexplored app states. A^3E [40], CuriousDroid [50], and Droidbot [37] dynamically generate finite-state models that capture transitions among activities to guide the exploration. Our approach differs from these approaches by generating a universal model from a multitude of apps that can be reused among apps and testing tools. With this difference, we can overcome two common setbacks of model-based approaches. First, we can interact with apps that dynamically load content—and, thus, cannot be thoroughly analyzed statically. Second, we avoid the costs of executing the static analysis for each app under test, which is a time-consuming task.

Systematic testing approaches employ various algorithms to test applications exhaustively or generate tests that trigger specific behaviors. EvoDroid [48] and Sapienz [49] use search-based algorithms—evolutionary or combined with random fuzzing—to improve test coverage. ACTEve [47] and IntelliDroid [46] apply concolic testing to generate feasible event sequences for Android apps to trigger specific behaviors. Our approach is not tailored to explore apps completely or to trigger specific behaviors. Targeting specific behaviors mostly demand static analysis to identify the path of the app's execution flow, which must be followed. Testing a real app completely frequently demands prohibitory long testing times. Our approach instead focuses on reducing the number of ineffective actions performed during testing, therefore reducing the overall test duration or allowing more functionality to be explored. This translates into improving the coverage of the explorations and speeding-up testing.

4.6 Lessons Learned

This chapter presented the first of our approaches to learning the language of apps. More specifically, it presented an approach to determine *which UI element to interact with* by guiding app testing towards UI elements that are most likely to be reactive.

It presented our approach to statically learn a *UI interaction model*, from a multitude of apps, which captures associations between UI elements and interactions. Test generators can consume the *UI interaction model* during dynamic explorations to predict UI elements with higher probabilities of success.

We applied the approach into two existing state of the art testing tools: DM-2 and DROID-BOT. Our experimental evaluations demonstrated the *applicability* and *efficacy* of our approach. The proposed guided exploration reduces the number of ineffective actions on the order of 50% and experiences an improvement in code coverage of up to 43%.

Our approach is orthogonal and complementary to current dynamic analysis and UI-exercising approaches. The UI model can be plugged into existing tools to improve effectiveness. The extension of the state of the art tools used in our evaluation only takes ~100 LOC.

This work opens new research directions that we would like to tackle in the future:

Event type prediction. While our approach demonstrates the benefits of identifying which widgets are more likely to possess an attached event handler, it did not determine which event should happen. This topic is addressed in our next chapter.

Hybrid and adaptive models. In Section 4.3.5, we illustrated the benefits and limitations of using a single-app model. Crowd knowledge could be combined with app specific knowledge or could be fine-tuned (adapt) during testing. In the next chapter, we explore *hybrid models* combining our crowd knowledge that combine crowd knowledge with app specific information.

Semantic data. Another further improvement is to enhance the model with textual and graphical semantics. Static analysis also provides texts and images. Without knowledge of the underlying app code, a user can recognize that a UI element with text *"Click here"* or with a "+"' icon should react to an *onClick* event.

Usability testing. Finally, we focused on improving *test generation*; however, the approach could be applied for *usability testing*. Interaction models can detect bad designs that do not follow the norm, leading to bad user experiences.

Reproducibility: To facilitate the reproducibility of experiments, the tools and dataset used in the evaluation are available online:

Droidmate-M and dataset: `https://github.com/uds-se`
Droidbot-M: `https://github.com/natanielrj/droidbot`

Chapter **5**

Learning Widget Interactions

This chapter is taken, directly or with minor modifications, from our 2019 ISSTA paper *Learning User Interface Interactions* [82]. My contribution in this work is as follows: (I) original idea of using reinforcement learning to learn app interactions; (II) original idea of using a statically mined UI model as previous knowledge; (III) idea and development of the fitness proportionate selection hybrid; (IV) partial evaluation.

Automated test generators systematically identify and interact with user interface elements to explore the functionality of a mobile app. One key challenge is to synthesize inputs which effectively and efficiently cover app behavior. In the previous chapter (Chapter 4), we explained how to learn interaction patterns from multiple apps and how to reuse them when testing a new app. This allowed us to generate more efficient tests by inferring which user interface elements should be interacted with. In many cases, however, it is not enough for a test generator to determine *which* element to interact with, but also *how* to interact with it.

As an example, consider Figure 21, showing a screenshot of the Android AAT activity tracking app. Already for humans, interacting with this screen can be quite a challenge—what exactly do the individual icons do? The text area in the white information box ("tourismcampsite") is scrollable; swiping on it scrolls the text, eventually revealing buttons at the bottom, which opens up further functionality. A test generator may click on random parts of the screen without prior knowledge, but randomly generating a series of swipes is unlikely to scroll the entire text.

Recent research attempts to emulate such knowledge for test generators by gathering knowledge from other apps and transferring this knowledge to new apps. Our crowd-model (Chapter 4) mined associations between UI elements and their interactions from popular apps and can learn, for instance, that buttons typically accept interactions Additionally, *dynamic techniques*, such as Korogulu's [83], learn these actions from dynamic executions and again apply a pre-approximated probability distribution mined from several apps. A limitation of both these approaches is that they are strongly *biased* towards the distribution initially mined. They work well if the app under test is similar to those used to train the model, but fail if it is dissimilar. Consider the the "tourismcampsite" widget to be swiped in Figure 21—a "layout" object:

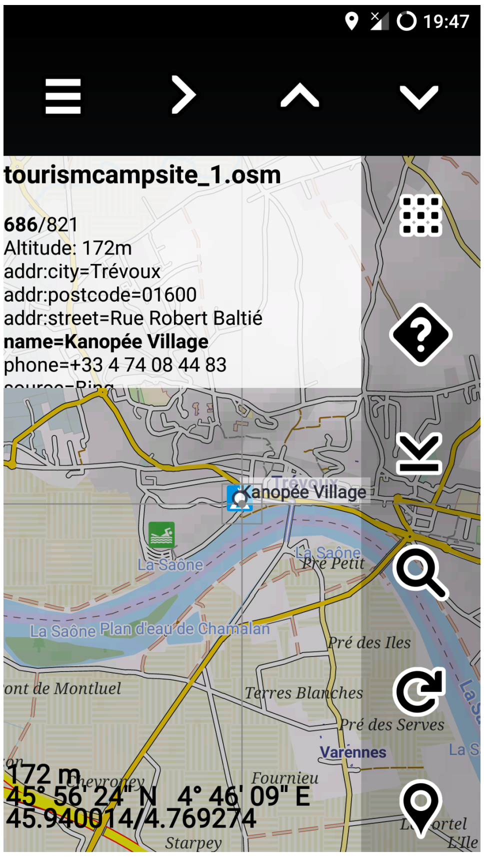


Figure 21: Map screen from the activity tracking app AAT app. The UI elements can be interacted with in different ways: Whereas the icons would react to *clicking* on them, the white information box ("tourismcampsite") must be *swiped* from bottom to top to have new UI elements at the bottom of the text scroll in.

- If the model is trained from apps that do not associate a *swipe event with layout objects*, test generators using the model would not swipe it, missing UI elements at the bottom.
- If the model is trained to swipe on *layout* elements, the test generator would also needlessly swipe on the top and right menu layouts, even if they do not react to swipes.

What is needed is a technique that automatically *adapts* the model to the app at hand, using a preconceived notion of *likely* interactions, while *gradually including less likely actions* if the former do not succeed.

To this end, we approach test generation as an instance of the *multi-(or N-) armed bandit problem* [84] (*MAB problem*). In this probability theory problem, a finite set of resources (actions) has to be distributed among competing alternatives (UI elements) to increase its reward (test quality). We use reinforcement learning to address test generation from this perspective and to systematically and gradually adjust our test generation strategy towards the application under test.

We organize the remained of this chapter according to our contributions. After introducing the multi-armed bandit problem and the techniques traditionally used to address it (Section 5.1), we introduce test generation as an instance of the *MAB problem* (Section 5.2), formulating two strategies for reinforcement learning without prior knowledge. We then show, in Section 5.3, how to enhance our reinforcement learning models with *statically trained models*—such as our previous *UI interaction model* from Section 4.1—to demonstrate how to integrate pre-approximated probability distributions with a dynamically adjusted model.

In Section 5.4, we *evaluate* both strategies and show that *reinforcement learning can be used to test apps more effectively.* Compared to a statically gathered crowd-model, the average coverage increases by more than 18%. We also show that *reinforcement learning without a priori knowledge outperformed those with a priori knowledge* by up to 8%, showing a clear advantage of the *MAB problem* approach over pre-mined models. As the last evaluation, we show that *reinforcement learning, added to a statically mined model, improves its coverage by an average of 20%.* After discussing limitations (Section 5.5) and related work (Section 5.6), Section 5.7 concludes the chapter.

5.1 The Multi-Armed Bandit Problem

Our premise in this work is that users learn not only how to test apps and reuse this knowledge when interacting with a new app. Instead, they actively adapt their behavior *while testing a new app.* That is, they *learn* how to interact with an app.

We approach test generation as an instance of the *MAB problem* [85]. This probability theory problem is illustrated as follows: given a set of competing slot machines (also known as one-armed bandits) to play, a player must decide which one to pull. Each machine has a different probability of generating a reward, and the player has no information about these probabilities. After each play, the player must decide if it will continue playing the same machine or change to another. After pulling an arm, it receives a reward based on the machine's probability distribution. By iteratively playing one machine at a time and observing the associated reward, the player can focus on the most rewarding machines, albeit with no knowledge about the actual probability distributions.

A *MAB problem* is formally equivalent to a one-state Markov decision process [86]. It can be defined as a tuple (A, R) where A is a set of $N \in \mathbb{N}^+$ possible actions, one for each arm, and $R(r|a)$ an unknown probability distribution of rewards. At each time step t the agent selects an action $a_t \in A$ and the environment generates a reward $r_t \sim R(\cdot, a)$. The agent's goal is to maximize the cumulative reward $\sum_{t=1}^{T} r_t$. If the reward of the *MAB problem* is either 1 or 0, it is called a *binary multi-armed bandit* or *Bernoulli (multi-armed) bandit* [87].

The *MAB problem* has been addressed with reinforcement learning techniques, such as *ϵ-greedy* [88] and *Thompson sampling* [89], with good results [90, 91].

The ϵ-GREEDY strategy, illustrated in Figure 22, considers a pre-defined threshold ϵ – in the interval $(0, 1)$ – to determine if it will *explore* new elements or *exploit* its current knowledge. For each action, this approach has a probability ϵ of randomly pulling an arm (exploration) and a probability of $1 - \epsilon$ of pulling the arm with the highest potential reward (exploitation).

Thompson Sampling—also known as posterior sampling or probability matching [92]—selects an arm by randomly sampling an estimate from each arm's posterior distribution and selecting the arm with the best sample. For Bernoulli bandits, i.e., those with a binary reward, this posterior distribution is a beta-distribution (B) with parameters α and β. It starts with an

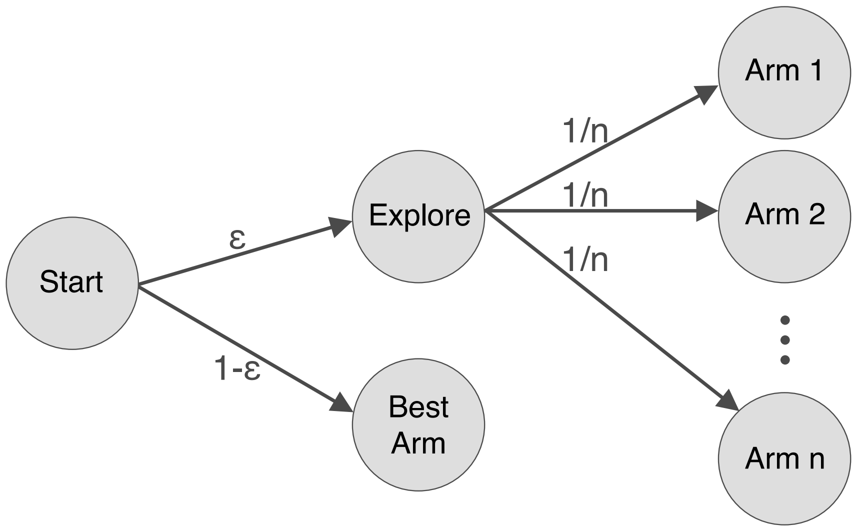
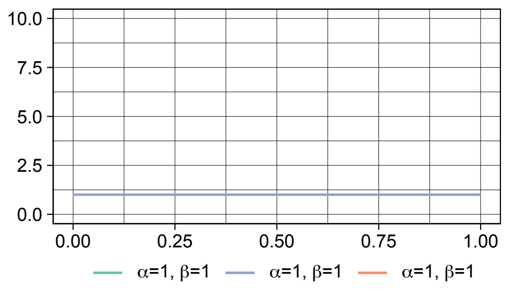
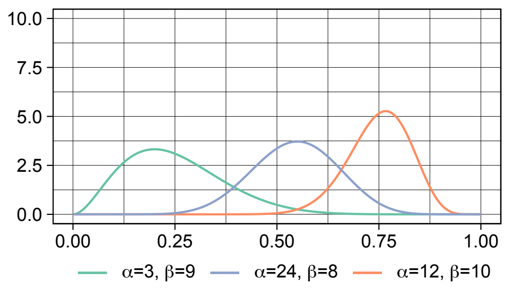


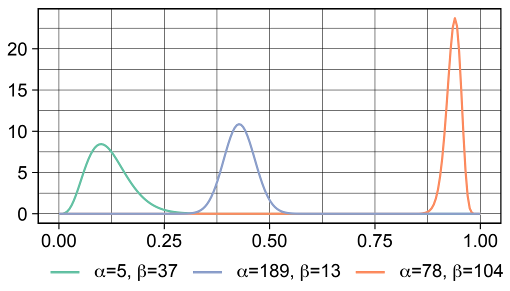
Figure 22: Overview of the ϵ-Greedy *approach.*



(a) Initial distribution



(b) Distribution after a few updates



(c) Distribution after some more updates

Figure 23: Three B-distributions changing over time

independent prior belief over each arm's mean reward ($\alpha = 1$ and $\beta = 1$, as $B(1, 1)$ is uniform distribution on $(0, 1)$), as illustrated in Figure 23a.

It then pulls an arm and updates the distribution parameters according to the reward. If the pull was successful ($r = 1$), *alpha* is increased by 1, and if it was not successful ($r = 0$), β is increased by 1. The distribution becomes more concentrated as $\alpha + \beta$ grows, as illustrated in Figure 23b and Figure 23c. The arms with higher mean rewards have a higher probability of their estimate being the best one and, therefore, are played more frequently (*exploitation*). Arms with a low mean reward, however, are not removed, but they are selected with a smaller frequency (*exploration*).

5.2 Testing With Reinforcement Learning

When testing an app, a test generator is presented with UI states containing multiple elements. It must choose not only which widget to interact with but also which type of interaction to perform. Each type of interaction on each UI element has its probability of improving testing. Our goal is to learn how to interact with an app dynamically, that is, to identify which UI elements should be triggered and which kind of interaction should be used, reducing the number of ineffective actions performed.

We model this problem as an instance of the *MAB problem*, in which the test generator action (resource) has to be allocated between widgets and action types (competing alternatives) to improve the tested behavior (reward). Based on our model, we implement two traditional reinforcement learning strategies to address it: ϵ-Greedy and *Thompson Sampling*.

5.2.1 Model Definition

The *MAB problem* is constituted of three major components: *arms*, *probabilities* and, *reward*.

Arms. The standard *MAB problem* definition supports only one type of action per arm. For test generation, we need multiple types of actions per arm, with independent probability distributions. Thus, we model each competing arm as a pair (w, a), where w is a widget, and a is an action type supported by the test generator. We denote as A the set of all interaction types supported by the test generator and as a state S the set of UI elements available on the app screen at a specific time. To pull an arm is equivalent to act a specified in the (w, a) pair on the widget w.

Probabilities. Based on our arm definition, we denote the probability of each action triggering an app response as $P(a|w) = p$, where p is the probability of the widget w reacting to the action a. To consider an independent probability for each UI element in the app would not allow knowledge to be transferred between UI elements, as the result of each action would be valid for a single widget. We, thus, cluster widgets into classes $C(w)$ and assign the probability to these classes instead of to individual widgets, that is, $P(a|w) \equiv P(a|C(w))$.

Based on the work from [68] we defined the widget classes as tuples $C(w) = (t_w, p_w, c1_w, c2_w)$. Where t_w is its class type, p_w is its parents class type and $c1_w$ and $c2_w$ are the class type of its first and second children.

Rewards. We determine our reward r as either 1 or 0, according to visual changes in the app.

$$r = \begin{cases} 1, & \text{if app's UI changed after executing } a \\ 0, & \text{otherwise} \end{cases}$$

We consider actions that trigger a visual change in the app as *effective* and those that do not as *ineffective*. Thus, while maximizing the overall reward, we aim to minimize the number of ineffective actions.

We opted to measure visual changes on the app UI to determine the action effectiveness not to restrict our approach to any specific apps or environment. We, therefore, consider an action *effective* if the screens before and after acting are different.

Our heuristic relies on design principles [80] that dictate there should always be a visual notification to the user after a reaction in the app. While this is an approximation, as an action could, for example, start a process in the background without notifying the user, actions that do not trigger UI changes are frequently classified as possible misbehaviors [79].

5.2.2 ϵ-Greedy Strategy

We modeled the ϵ-Greedy approach according to Algorithm 1. We first initialize the current (*wins*) and total (*trials*) counters and probabilities for all classes C and action types A (line 2-5). Since our algorithm starts with no previous app knowledge, we initialize all *wins* and *trials* with 0.

Until all resources are used, we obtain the expected reward of all elements on the app's current screen (state) and draw a random number (lines 6-8) to decide whether we select a random widget (line 9) or select the one which has the highest reward probability, given the current knowledge[18] (line 11). We then perform the action a on the widget w, obtain a reward r (line 13) and use this reward to update the counters (*trials*, *wins*) and the probability for the class (line 14-16). The counters *wins* and *trials* for each action type and class represent our accumulated knowledge.

Algorithm 1 ϵ-Greedy approach. Before each action it selects between acquiring new knowledge (*exploring*) or exploiting the best widget-action on the screen, based on its current knowledge (*exploitation*)

```
Require: $\epsilon \in (0,1)$
 1: function $\epsilon$-Greedy
 2:     for $(a,c) \exists A \cdot C$ do
 3:         $wins_{(a,c)},\ trials_{(a,c)} \leftarrow$ knowledge-base$(a,c)$
 4:         $P(a|C(w)) \leftarrow \frac{wins_{(a,C(w)}}{trials_{(a,C(w)}}$
 5:     end for
 6:     while stop criteria not met do
 7:         $S = (w, P(a|w)) \forall\ w$ in the current screen
 8:         if $random() > \epsilon$ then
 9:             $e \leftarrow random\ w \in S$
10:         else
11:             $e \leftarrow \max(P(a|w)), (w, P(a|w)) \in S$
12:         end if
13:         $r \leftarrow e.w.$perform$(e.a)$
14:         $wins_{(a,C(w))} \leftarrow wins\ +r$
15:         $trials_{(a,C(w))} \leftarrow trials\ +1$
16:         $P(a|C(w)) \leftarrow \frac{wins_{(a,C(w)}}{trials_{(a,C(w)}}$
17:     end while
18: end function
```

[18]If two or more widgets have the same probability, we randomly select one.

5.2.3 A Thompson Sampling Based Strategy

Our *Thompson sampling* approach, shown in Algorithm 2, starts analogously to the ϵ-Greedy one, by initializing the (*wins*) and total (*trials*) counters and probabilities for all classes C and action types A (line 2-5). In contrast to the ϵ-Greedy approach, however, the probability is a B distribution, based on the wins and trials.

Until all resources are used, we sample the probability distribution for all elements on the app's current screen (state) and select the best sample, that is, the one with the highest value (lines 6-9). We then perform the action a on the widget w, obtain a reward r (line 10) and use this reward to update the counters (*trials*, *wins*) and the probability distribution for the class (line 11-13). The counters *wins* and *trials* for each action type and class are used to recalibrate our probabilities after performing each action, and the β distributions encode the accumulated knowledge.

Algorithm 2 Thompson Sampling based approach for test generation. For all UI elements on the screen, a sample is taken from a B distribution and the UI element with best sample is selected

```
1:  function THOMPSON-SAMPLING
2:      for $(a, c) \exists A \cdot C$ do
3:          $wins_{(a,c)}$, $trials_{(a,c)} \leftarrow$ knowledge-base$(a, c)$
4:          $P(a|C(w)) \leftarrow B(1{+}wins_{(a,c)}, 1{+}trials_{(a,c)}{-}wins_{(a,c)})$
5:      end for
6:      while stop criteria not met do
7:          $S \leftarrow P(a|C(w)) \forall\ w$ in the current screen
8:          $L \leftarrow sample(s) \mid \forall\ s \exists S$
9:          $e \leftarrow \max(L)$
10:         $r \leftarrow e.w.\text{perform}(e.a)$
11:         $wins_{(a,C(w))} \leftarrow wins + r$
12:         $trials_{(a,C(w))} \leftarrow trials + 1$
13:         $P(a|C(w)) \leftarrow B(1{+}wins_{C(w)}, 1{+}trials_{C(w)}{-}wins_{C(w)})$
14:     end while
15: end function
```

5.3 Reinforcement Learning With Previous Knowledge

When users interact with an app, they not only learn while using it but also reuse their previous knowledge about how to use apps. Similarly, our techniques support the use of a priori knowledge alongside reinforcement learning. Also, reinforcement learning can be used alongside previously existing approaches to enhance previously gathered models with information about the app under test.

5.3.1 Integrating Static Models

Our reinforcement learning techniques support the reuse of previous knowledge through the *knowledge-base* function (Line 3) in Algorithm 1 and Algorithm 2.

We modeled the use of a priori data according to Algorithm 3. We obtain the probability values for the action and class (line 2) from the a priori knowledge. We then initialize the number of wins with the value obtained from the previous knowledge, weighted by ψ, and the number of trials as ψ. Higher ψ values give more weight to the previous knowledge and make the newly acquired knowledge to have a smaller initial effect.

Algorithm 3 Knowledge-base function to reuse a priori knowledge alongside our ϵ-Greedy and Thompson sampling approaches

```
1: function KNOWLEDGE-BASE(action, class)
2:     $p \leftarrow$ probability($action, class$)
3:     $wins \leftarrow p \times \psi$
4:     $trials \leftarrow \psi$
5:     return wins, trials
6: end function
```

5.3.2 Extending Exploration Strategies

The principles of reinforcement learning can be applied to other test generation strategies for more effective testing. To illustrate the benefits of using reinforcement learning on different test generation approaches, we propose an extension to the *crowd-based dynamic exploration* [68]. We extend the authors' original approach with our reinforcement learning model so that the test generator can adapt to cases where the original model was not so useful.

Our extended algorithm is shown in Algorithm 4. We first initialize our trials and wins for all classes C and action types A (lines 2-7) with the values from the original *crowd-based model.* We then execute the exploration until a stop condition is met (Line 8). To generate each interaction, we trigger our original stochastic select algorithm (4) while replacing the *crowd-model* for our extended version (Line 10).

We then perform a on the widget w and obtain a reward r (Line 11) and update the *trials* and *wins* counters and the class probability, according to the action result (Lines 12–14). In this algorithm, the *wins* and *trials* counters for each action type and class allow the *crowd-based* model to adapt according to the app behavior.

We implemented our approach as plug-ins for DM-2 [53]. Since Android does not provide a unique identifier for UI elements, we relied on DM-2 heuristics, based on their textual or graphical content, to uniquely identify them. It then uses these UI elements as a heuristic to uniquely identify a UI state. We reuse this metric to determine if an action was effective; therefore, if the UI state before the action is different from the state after, we consider that the action was effective.

We extended the original set of capabilities from DM-2. In its original version, it only performs clicks and long clicks to trigger app behavior. We included four swipe events: swipe up, down, left, and right, for scrollable widgets – according to their Android properties.

5.4 Evaluation

Our approach aims to learn to interact with an app while testing it. We, thus, conducted a set of experiments to measure its effectiveness and benefits. In the remainder of this section, we present our dataset and our answers to the following research questions:

Algorithm 4 Fitness Proportionate Selection with reinforcement learning. Starting from the widget class probabilities from the crowd-based model and dynamically tuning these values according to the behavior of the app under testing.

```
Require: $n > 0$
1:  function FITNESS-PROPORTIONATE-SELECTION-WITH-REINFORCEMENT LEARNING
2:      for $(a, c) \exists A \cdot C$ do
3:          $p \leftarrow$ crowd-based-model($action, class$)
4:          $wins \leftarrow p \times \psi$
5:          $trials \leftarrow \psi$
6:          $P(a|C(w)) \leftarrow \frac{wins_{(a,C(w)}}{trials_{(a,C(w)}}$
7:      end for
8:      while stop criteria not met do
9:          $S = (w, P(a|w)) \forall w$ in the current screen
10:         $e \leftarrow$ originalStochasticSelectAlgorithm($S$, $P$)
11:         $r \leftarrow e.w$.perform($e.a$)
12:         $wins_{(a,C(w))} \leftarrow wins + r$
13:         $trials_{(a,C(w))} \leftarrow trials + 1$
14:         $P(a|C(w)) \leftarrow \frac{wins_{(a,C(w)}}{trials_{(a,C(w)}}$
15:     end while
16: end function
```

RQ1 (Learning Without a Priori Knowledge). Can reinforcement learning be used to more effectively test apps?

RQ2 (Learning With a Priori Knowledge). Is knowledge learned from the app under test more beneficial to testing than static models from other apps?

RQ3 (Extending Static Models). Can reinforcement learning be used to enhance static models?

5.4.1 Experimental Setup

We reused the set of benchmark apps from [68], shown in Table 6, as well as the number of statements, widgets, and events found by static analysis in their experiment. We obtained the same app versions used in the original experiments from the *Google Play Store*[19], the official market for Android apps, and from *F-droid*[20], an open-source repository of Android apps. Compared to the original work, we excluded the apps Alogblog, Jamendo, DroidWeight, Tomdroid, and SyncMyPix, because they either no longer work on newer versions of Android or were unavailable for download.

5.4.2 RQ1: Learning Without A Priori Knowledge

By guiding test generation towards more effective UI elements, we previously demonstrated (Chapter 4) that we could cover parts of the application faster. In this experiment, we want to

[19]`https://play.google.com/store/apps`
[20]`https://f-droid.org`
[21]This app crashed when being evaluated with BACKSTAGE.

Table 6: Set of evaluation benchmark apps

App	**Source**	**#Stmts**	**Widgets**	**Events**
KeePassDroid (2.0.6.4)	F-Droid	43	169	0
Munch (0.44)	F-Droid	8084	387	0
BART Runner (2.2.6)	F-Droid	8125	170	5
2048 (2.06)	F-Droid	168	3	1
Pizza Cost (1.05-9)[21]	F-Droid	1240	N/A	N/A
Mirrored (0.2.9)	F-Droid	2475	29	0
Easy xkcd (5.3.9)	F-Droid	13768	265	6
Dialer2 (2.90)	F-Droid	2005	55	19
PasswordMaker (1.1.11)	F-Droid	4378	177	30
World Weather (1.2.4)	Play Store	4116	205	0
Der Die Das (16.04.2016)	Play Store	3225	69	0
wikiHow (2.7.3)	Play Store	3703	183	7

gather empirical evidence that it is possible to achieve a similar outcome without collecting data a priori, but instead by learning how to use an app effectively while doing so.

With this goal, we compared our implementations of the ϵ-greedy and Thompson Sampling strategies (henceforth ϵ-Greedy and Thompson) against Borges *et al.*'s crowd-based approach (henceforth Baseline). We compared only against this implementation as it has been shown in previous research [53, 68] to outperform DroidBot [37], Monkey [30] and the original DroidMate on this same dataset.

We explored each of the 12 apps from our test dataset ten times on Google Pixel 2 XL devices running Android 8.1 (API 27), and we obtained the average coverage from these tests. We again opted for ten runs per app to mitigate the noise caused by the semi-random search and app non-determinism. In each run, we programmed the test generator to trigger 1000 actions, including an app restart, after every 100 actions to increase the probability of exploring different app branches. We opted for 1000 actions as it represents $\approx$ 20 minutes of exploration, and more than 15 minutes have been shown not significantly to increase the coverage of random testing tools [53]. Moreover, we used the same amount of actions for both approaches since the reinforcement learning approaches do not have a meaningful performance penalty compared to the baseline.

5.4.2.1 Parameter calibration

Our ϵ-Greedy approach requires the value of ϵ to be determined. This value is used to determine the strategy exploration/exploitation rate and has a significant impact on its behavior. Before our experiment, we performed a small-scale experiment to determine its value. We randomly selected five apps from the test dataset and explored them 4 times for each of the following ϵ values: 0.05, 0.1, 0.2, and 0.3. Based on the coverage variation, we opted for $\epsilon = 0.3$. The results of these experiments are shown in Appendix C.

5.4.2.2 Results

The results of this experiment are shown in Figure 24. ϵ-Greedy performed better than the Baseline, achieving ≈18% more coverage on average. Thompson outperformed both strategies, achieving ≈24% more coverage than the Baseline and 6% more than ϵ-Greedy. On a per-app analysis, presented in Figure 25, Thompson obtains more coverage on eight apps, ϵ-Greedy obtains more coverage on on,e and Baseline obtains more coverage in 3. To verify the statistical significance of our results, we performed a Friedman's test. It resulted in a *p-value* < 0.00001, indicating that our results are significant at 5%.

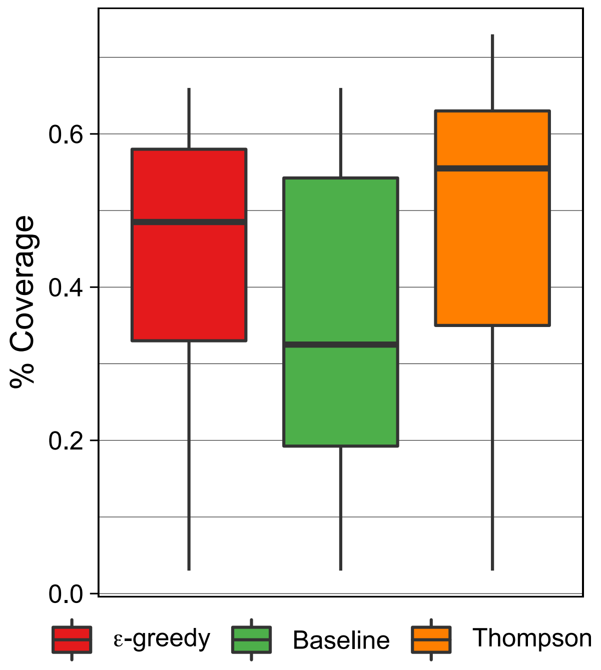


Figure 24: Comparison of statement coverage between Baseline, ϵ-Greedy, and Thompson

ϵ-Greedy and Thompson strategies led to an average coverage increase of 18% and 24%, respectively, when compared to a statically gathered crowd-model.

5.4.3 RQ2: Learning With A Priori Knowledge

Our previous experiment showed that reinforcement learning approaches could guide test generation towards more effective UI elements, leading to better test coverage. In this experiment, we want to gather empirical evidence that information gathered through reinforcement learning is more beneficial to the test result than the information gathered statically and reapplied.

With this goal, we compared our implementations of the ϵ-greedy and Thompson sampling strategies, started with a priori knowledge (henceforth ϵ-Greedy+K and Thompson+K) against their counterparts with no starting knowledge. Similarly to RQ1, we explored each app from our test dataset ten times – to mitigate noise – on Google Pixel 2 XL devices running Android 8.1, and we obtained the average coverage from these tests. In each run, we configured

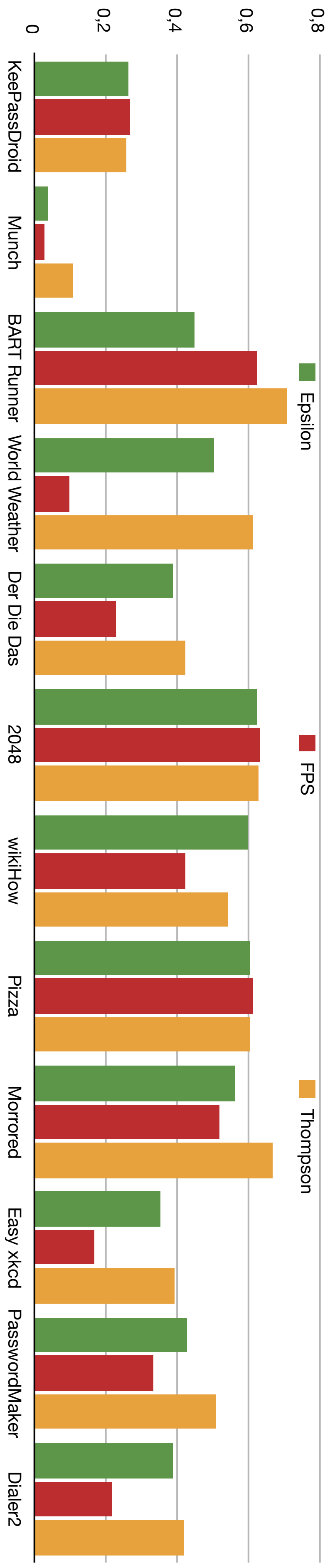


Figure 25: Per app comparison of statement coverage between Baseline, ϵ-Greedy, and Thompson

the test generator to trigger 1000 actions, including an app restart, after every 100 actions to increase the probability of exploring different app branches. We used the UI interaction model mined and trained by [68] as a priori knowledge, since it is representative of app behavior and, similar to our approach, can be used on any arbitrary app.

5.4.3.1 Parameter Calibration

To use a priori knowledge alongside our reinforcement learning approaches, we need to define a weight ψ for the model. Higher values of ψ reduce the reinforcement learning effect and increase the relevance of initial knowledge, and smaller values give a higher significance to knowledge gained through reinforcement learning. To determine a value for ψ, we randomly selected five apps from the test dataset and explored them four times, using Thompson+K, for each of the following ψ values: 10, 20, 50, and 100. Based on coverage obtained by these tests, we opted for $\psi = 20$. The results of these experiments are shown in Appendix C.

5.4.3.2 Results

The results of our comparison between ϵ-Greedy, ϵ-Greedy+K, Thompson, and Thompson+K are shown in Figure 26. Both ϵ-Greedy+K and Thompson+K achieved a lower average coverage than their counterparts without a priori knowledge—4% and 8%, respectively. This indicates that the knowledge obtained during the testing is more valuable than the knowledge from the model. A priori knowledge prevented bad random seeds from achieving especially bad results during testing, increasing the minimum overall coverage achieved. On a per-app analysis, ϵ-Greedy obtains more coverage on two apps, ϵ-Greedy+K obtains more coverage on four apps–including three draws against Thompson or Thompson+K. Thompson obtains more coverage on eight apps–including two draws against ϵ-Greedy+K–, and Thompson+K obtains more coverage on one app–drawing against ϵ-Greedy+K. To verify the statistical significance of our results, we performed a Friedman's test. It resulted in a *p-value* of 0.00073, indicating that our results are significant at 5%. The individual app results are shown in Appendix C.

Reinforcement learning without a priori knowledge outperformed those with a priori knowledge by up to 8%, indicating that reinforcement learning data is more beneficial to testing than statically gathered data.

5.4.4 RQ3: Extending Static Models

Our previous experiment indicated that the knowledge gathered by reinforcement learning while testing an app is more relevant to the test quality than a priori knowledge. In this experiment, we gather empirical evidence that other test generation approaches can benefit from the use of reinforcement learning.

For this experiment, we extended the Baseline algorithm, allowing it to adjust its knowledge through reinforcement learning while testing an app. We denote this extension as Baseline+K. Similarly to the previous experiments, we explored each app from our test dataset ten times—to mitigate noise—on Google Pixel 2 XL devices running Android 8.1, and we obtained the average

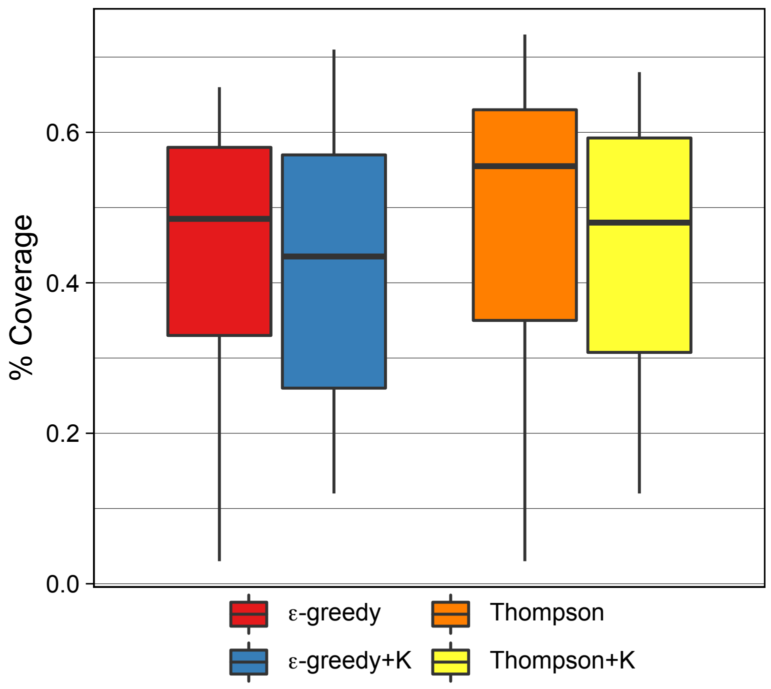


Figure 26: Comparison of coverage between ϵ-Greedy, ϵ-Greedy+K, Thompson and Thompson+K.

coverage from these tests. In each run, we programmed the test generator to trigger 1000 actions, including an app restart after every 100 actions to increase the probability of exploring different app branches.

The results of our comparison between Baseline and Baseline+K are shown in Figure 27. Baseline+K performed significantly better than the Baseline– with a 20% coverage increase. These results also indicate that knowledge obtained during the testing is more valuable than the a priori knowledge from the model, as the Baseline does not obtain knowledge from the app under test during testing, but Baseline+K does. On a per-app analysis, Baseline obtains more coverage on two apps, and Baseline+K obtains more coverage on the remaining ten apps. To verify the statistical significance of our results, we performed a Friedman's test. It resulted in a *p-value* of 0.02387, indicating that our results are significant at 5%. The individual app results are shown in Appendix C.

The addition of reinforcement learning to a statically mined model lead to 20% coverage improvement.

5.4.5 Threats to Validity

Regarding external validity, we cannot ensure that our results generalize to all apps and testing tools, due to the size of our dataset. To mitigate this threat, our benchmark apps were taken from previous research, which sampled from a variety of sources, commercial (Google Play Store) and open-source (F-Droid); and from a variety of different categories. We used Android due to its popularity, but the concepts presented in this paper also apply to other platforms or domains

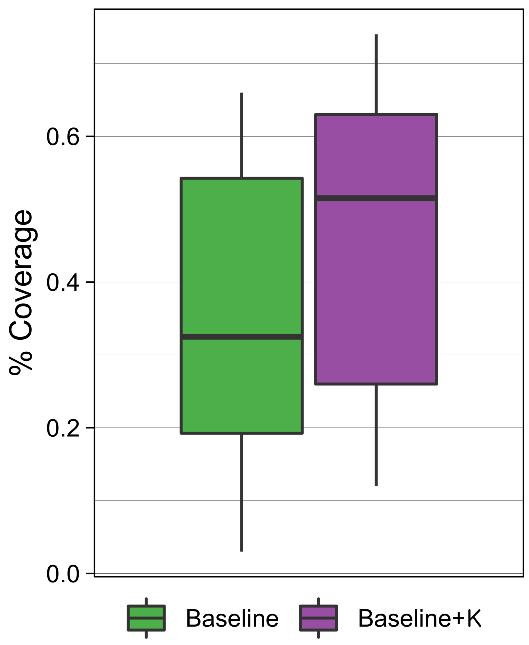


Figure 27: Comparison of coverage between Baseline and Baseline+K

(e.g., iOS and web applications). For further confidence in external validity, more evaluations on other tools and platforms are necessary.

Regarding construct validity, our approach is implemented on top of DM-2 and inherits its limitations. Apps that do not work on DM-2 do not work on our approach. Our model and strategy are, however, generic and could be adapted for any other tool. An additional construct limitation is the use of random testing. Our approach comprises prioritizing test inputs, instead of randomly interacting with them. However, it does not analyze the semantics of a UI to input meaningful values on input fields or perform tasks in a human-like way. Therefore, our approaches' maximum coverage is limited by the inherent limitations of random exploration strategies. However, our ϵ-Greedy and Thompson approaches could be applied alongside any systematic or model-based testing tool to assist them in prioritizing between alternatives. Finally, to exploit a priori knowledge, we used the UI interaction model we mined and trained in Chapter 4; thus, this data suffers from the same threats to construct validity we previoulsy presented: the extracted information for the UI interaction models is incomplete because of the inherent limitations of static analysis and the actual tool, *Backstage* [74], used to mine them. Our approaches, however, do not require a priori knowledge.

Regarding internal validity, we only instrumented Java byte code to measure the coverage and, therefore, not measured the coverage of other parts of the app code, such as web content and native code. While there is a strong correlation between finding faults and the code coverage of a test suite, the use of a curated repository of bugs could provide more accurate results regarding the techniques' effectiveness.

Finally, the measurement of effective actions (UI change) is a heuristic based on design guidelines and usability principles. If an app does not follow these principles, the heuristic may not hold.

5.5 Limitations

Our experiments showed the benefits of learning or refining an interaction model while testing apps. Nevertheless, our implementation and approach would not work for all apps.

Our implementation inherits all limitations from DM-2, which we previously discussed in Section 3.7. Moreover, in our approach, similar limitations to those from the statically mined interaction model (Section 4.4), as both approaches are conceptually similar.

In addition to these limitations, our approach is unable to determine how effective an action is concerning the amount of functionality they trigger. Instead, we rely on a simple image comparison algorithm (subtraction) to measure the effectiveness of an action. While this approach is fast to execute, it is susceptible to UI changes. If a widget moves a few pixels or has a different graphical representation when focused, we consider the action as effective, even if it has not triggered any app behavior. One could imagine that the code triggered by an action could be a more accurate measurement of effectiveness. However, such a metric would not work for components such as checkbox and radio groups, which frequently do not directly trigger app behavior. Being unable to determine how effective an action was accurate, limited how we modeled our reinforcement learning approach. We opted to model the *MAB problem* using *Bernoulli multi-armed bandits*, i.e., a model with a binary reward. Determining how to quantify app functionality and model it as a reward guide test generation remains an open research question.

5.6 Related Work

This section summarizes the most relevant works related to this research.

The closest work to the approach we presented in this chapter is iMPAcT [39], which identifies the UI patterns used in each app screen and compare them to an internal catalog of patterns to determine how to interact with the app. We, similarly, also learn how to interact with an app. However, we do it entirely dynamically, without relying on a human-defined catalog of interaction patterns. Moreover, we learn how to interact with the specific app that is being tested; thus, our approach works even if the app under test requires different types of interactions as other apps.

Similarly, other test generation approaches have also experimented with reinforcement learning. *GUI testing with reinforcement learning* [93, 94] showed that reinforcement learning could be used for automated GUI (robustness) testing. As a reinforcement learning algorithm, they used *Q-learning* [95], a popular model-free reinforcement learning technique. Esparcia *et al.* [96] also used *Q-learning* as a meta-heuristic for action selection in their testing tool and showed that the superiority of action selection by *Q-learning* could only be achieved through an adequate choice of parameters. Koroglu *et al.* [83] used *Q-learning* for Android GUI testing to achieve activity coverage and to detect crashes. They used *offline Q-learning*—split into a learning phase and a testing phase: During the learning phase, their approach learns an abstract model from multiple apps. It then uses the gained knowledge (Q-matrix) as a model to predict which actions might lead to new activity functionality or a crash. Their approach uses a reinforcement learning technique, but is closer to our *UI interaction model* (Chapter 4) than to our approach, as both learn static models. Our approach gains and applies knowledge during testing, generating a model that is specific to the app under test. Finally, our approach is closely related to

AUTOBLACKTEST [97]. It uses Q-learning to learn how to interact with each interaction type of each UI element in the app. Instead of learning how to interact with individual app elements app, our approach learns how to interact with classes of elements, allowing the knowledge to be transferred between different UI elements. Moreover, by using classes, we prevent a state explosion problem: there are only a few thousand possible combinations of widget classes and interactions in all apps. At the same time, there is a virtually infinite combination of states, UI elements, and interactions. Finally, by using classes of UI elements we can reuse a model learned on one app while testing different apps, similarly to how we reused our *UI interaction model* from Chapter 4.

In Section 2.2, we presented existing test generation approaches, which are loosely related to our work. Such test generators can benefit from knowing how to interact with UI elements. *Random testing* techniques, such as MONKEY [30], DYNODROID [31], and DM-2 [34], create random sequences chains of events to explore an app. Our approach can be used alongside such test generators to guide them towards events that are more likely to affect the app.

Model-based testing techniques infer app models using static, dynamic, or hybrid analysis. GUIRIPPER [35] and MOBIGUITTAR [36] create a state machine model of the app while testing it. ORBIT [42] follows a similar approach but uses static analysis to reduce the number of GUI elements to test. SMARTDROID [43] relies only on statically generated activity and function call graphs to identify paths that should be explored. Our approach can be used alongside such techniques. By learning which interactions to do, approaches that build an app model dynamically can better decide which actions to take. When they use static or hybrid analysis, our approach can help them handle unknown states or decide between two equally good alternatives.

Systematic testing approaches employ various algorithms to test applications exhaustively or to generate tests that trigger specific behaviors. For example, EVODROID [48] and SAPIENZ [49] use search-based algorithms, while ACTEVE [47] and INTELLIDROID [46] rely on concolic execution, and CRASHSCOPE [52] follows a depth-first-search approach. For such approaches, knowing which actions are more likely to work can be beneficial to guide test generation or to prioritize inputs to test.

5.7 Lessons Learned

This chapter presented the second of our approaches to learning the language of apps. More specifically, it presented our approach to addressing the problem of determining *how to interact with the UI element* through the use of reinforcement learning.

In this chapter, we modeled tests as an instance of the *MAB problem*, and we showed how techniques used to solve the *MAB problem* perform when used to generate tests. We also showed how o integrate our reinforcement learning approach with previously mined models, such as our *UI interaction models* (Chapter 4). Our results showed that reinforcement learning approaches—without previous knowledge—lead to an average code coverage improvement of 20% compared to a statically mined model.

Further use case scenarios might include saving the knowledge obtained during exploration in a refined model. Because of the fine adjustments of the model, their effects are more visible over time; thus, we expect that each new test can lead to a more efficient one. This refined model could, for instance, test new versions of the same app, allowing for a "continuous learning and testing" cycle. Moreover, it would also be possible to transfer refined models between different apps.

This work is the first approach to combine a crowd-based model with reinforcement learning exploration strategies and does not, by any means, cover the whole field. There is still much room for improvements and future work:

Coverage as a reward. Instead of using the action effectiveness of with a binary reward, one could use *coverage as a reward*, thus leading towards actions that explore more code locations.

Automated parameter calibration. Our preliminary experiments also showed that the values for ϵ and ψ significantly affect the exploration effectiveness and coverage. Our values were, however, selected based on a manually performed optimization. More adequate parameter values can be found through an *automated multivariate optimization experiment.*

Learning Rate. One could also introduce a *learning rate* α to our approach:

$$\begin{aligned} \mathit{trials}_{(a,C(w))} &\leftarrow \alpha \times \mathit{trials} + 1 \\ \mathit{wins}_{(a,C(w))} &\leftarrow \alpha \times \mathit{wins} + r \end{aligned}$$

The learning rate adjusts whether the algorithm should forget previous results quicker ($\alpha < 1$) and have a downward pressure toward ignorance, or whether the algorithm should act riskier ($\alpha > 1$) and be more resistant to changing environments.

Reproducibility: To facilitate replication and extension, all our work is available as open source. The replication package is available at:

`https://github.com/uds-se/droidmate-bandits`

CHAPTER 6

Learning Textual Inputs

This chapter is an expanded version of our 2020 AST submission *Testing Apps With Real-World Inputs [98]*. My contribution in this work is as follows: (I) original idea; (II) theoretical foundation; (III) partial implementation; (IV) partial evaluation.

Mobile applications (apps) that take complex data as input, such as travel bookings, maps, or online banking forms, are part of our everyday life. It does not suffice to know which UI elements to interact with to trigger their functionality. Instead, they require realistic and coherent test inputs to be tested adequately.

Consider the example in Figure 28. To search online for a book, the user is required to type an author, title, or ISBN. While it is possible to input virtually any value for author and title, the ISBN must contain only numbers and must be 10 or 13 digits long to bypass syntactic validation.

Now consider an *automated test generator* being used to test this app. To explore an app's functionality, a test generator would systematically identify the user interface elements and interact with them. However, to find a book (and to explore the functionality associated with having found a book), it must first either input an existing ISBN or a valid combination of author name and book title. However, random generated author and book names, or ISBNs, are unlikely to produce any results and, instead, fail to reach code regions located beyond input validation checks.

Even if the test generator were able to bypass the syntax validation rules, the generated values would rarely be semantically meaningful. Although the ISBN may be valid in itself, a random valid ISBN still is unlikely to point to some existing book.

Currently, these scenarios are handled by using a *curated set of inputs*, such as dictionaries, or by manually written values for specific inputs. Both approaches are laborious, expensive, and subject to human bias, undermining two of the main benefits of automated testing.

The past research of Mariani et al. [14] indicated that *knowledge bases* could be a reliable source of semantically coherent inputs. Their LINK tool would query the *DBPedia* data collection to identify data to be used in the tests. They then manually used the extracted data to generate complex system test inputs. If a field required a "ZIP" code, for instance, LINK would query for "ZIP" codes from *DBPedia*.

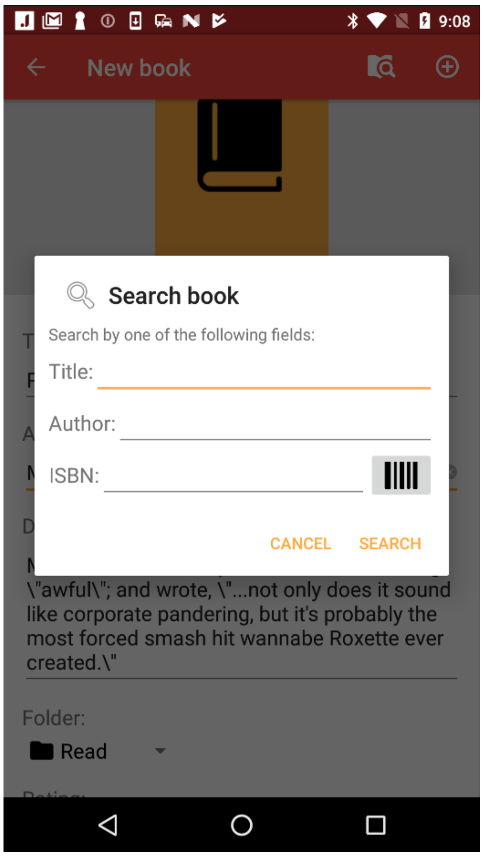

9:08
New book
Search book
Search by one of the following fields:
Title:
Author:
ISBN:
CANCEL
SEARCH
\"awful\"; and wrote, \"...not only does it sound like corporate pandering, but it's probably the most forced smash hit wannabe Roxette ever created.\"
Folder:
Read

Figure 28: Book search functionality in an app. It requires syntactically and/or semantically correct values to be tested

When LINK was conceived, it targeted desktop and web applications. However, would such an approach also work for mobile devices? Due to the limited screen size, mobile apps have different UI design patterns [38, 99], which diverge from those used on desktops, making a LINK-like approach less accurate. Nevertheless, LINK's core strategy to *query a knowledge base for input values* might be applicable for mobile devices too.

In this chapter, we investigate whether a LINK-like approach would help to generate better tests for Android apps, and if so, in what way. Our approach associate labels with input fields and extract its semantics; it then uses these concepts to consume data from a knowledge base and inputs these values during test generation in a user-like order.

The remainder of this chapter is organized according to our contributions. After discussing the approaches and tools used in this work (Section 6.1), we present a *set of metrics* that effectively associate descriptive elements with input fields, tailored to Android-specific design guidelines, and thus *extend* LINK *for use in mobile test generation* (Section 6.2).

We then evaluate our approach (Section 6.3). Our result shows that *concepts can be associated with input fields* with 87% precision. They also show that about *three out of four queries to the knowledge base returned valid results*, 94% of which were semantically valid, leading to *an average improvement of 10% in statement coverage* compared to random tests.

After discussing limitations (Section 6.4) as well as related work (Section 6.5), Section 6.6 prescribes future work and concludes our paper.

6.1 Background: Exploiting the Web of Data to Generate Inputs

Mariani *et al.* [14] proposed LINK to query input values from a knowledge base. Their work exploited the metrics based on the Gestalt principles [100] of how humans perceive objects and patterns to associate descriptor labels to input fields. In this section, we describe the underlying principles or their approach, which we adapted to the peculiarities of mobile apps.

6.1.1 Associating Descriptor Labels to Input Fields

LINK relied on the Gestalt principles of visual perception to associate descriptor labels with input fields on desktop GUIs. It used the metrics of *Proximity*, *Homogeneity*, and *Closure*, as implemented by Becce *et al.* [101]. These metrics work as follows:

Proximity Humans associate elements which are close to each other. Besides, app UIs are developed to be explored from left to right and top to bottom. Therefore, this metric dictates that descriptive labels must be located on the left or top of an input field.

Homogeneity Elements should be distributed according to their semantics. On UI design, the regular distribution of UI elements is mostly done through their alignment. This metric dictates that a label should be either vertically or horizontally aligned to the input field.

Closure Semantically correlated elements should be grouped for easier comprehension. Therefore, this metric dictates that semantically correlated UI elements should be placed in the same container.

6.1.2 Querying Knowledge Bases

Linked Data [102, 103] is a type of knowledge base which describes how to define and publish machine-readable typed links between arbitrary items on the Web so that it is interlinked and accessible through semantic queries. It is used extensively in different topical domains, including Media, Government, and Publications [104]. It builds upon standard Web technologies such as HTTP, Resource Description Framework (RDF), and Uniform Resource Identifiers (URIs). Links are represented with the RDF language, and URIs [105] identify the items. For example, the URI `http://dbpedia .org/resource/London` identifies the city of London, while the URI `http:// dbpedia.org/resource/Bird` identifies the animal category "Bird".

LINK exploited this structure to query for complex input values, based on the UI semantics while maintaining the semantic coherence between the inputs. It first queries DBPedia [106] classes and predicates for resource URIs whose name match those obtained in the label matching step. LINK uses WordNet to search for synonyms when it cannot find any class or predicate with the exact word queried.

It then associates the discovered elements by systematically querying for resources that occur as the subject of both elements. If a resource exists, the elements are merged into a single query. Otherwise, they are kept disjoint. Finally, LINK queries DBPedia for resources to obtain resources.

6.2 Method

We propose an approach with four steps, namely: *description matching*, *concept extraction*, *input value acquisition* and *input value consumption*, as illustrated in Figure 29.

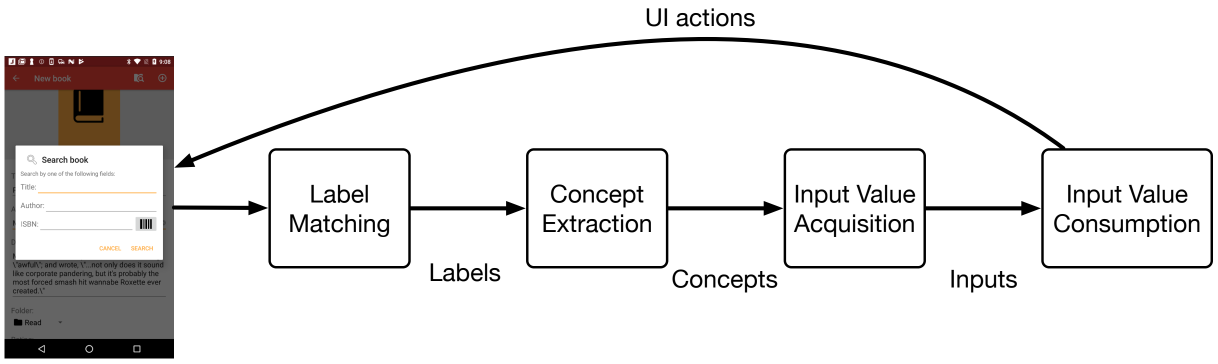


Figure 29: Approach overview diagram. Associate input fields and labels elements, extract the label's concepts and query for input values. Finally, use queried values to fill input fields during testing

Given a UI state, we start by identifying and *matching descriptive labels* with input fields, using a modified version of Becce's metrics adapted to mobile apps. We then use natural language processing (NLP) techniques [107] to *extract the concept* associated with the label. We use the extracted concepts, instead of the original labels, to *query* knowledge bases for input values. Finally, we fill all *input* elements with the queried values and randomly interact with the non-input elements.

6.2.1 Matching Labels

To match descriptor labels with input fields, we extend Becce's metrics to support mobile apps. We reuse their metrics of *Proximity*, *Homogeneity*, and *Closure* on the Android Window Hierarchy dump [108], which is similar to the DOM structure used in Becce's original work.

Becce's metrics are based on the idea that descriptor labels and input fields are distinct elements. This approach works on web and desktop apps because the hint text is frequently used to exemplify or assist the user in filling the input field, not to describe its meaning. This does not hold to mobile apps. Due to limited screen size, mobile apps frequently reuse the input element for descriptive proposes, through its hint text, as shown in Figure 30.

We thus add an *Enclosure* metric, which combines the Gestalt principles of *proximity* and *closure*. Our metric is defined as follow:

Enclosure Input fields can describe themselves to mitigate UI space requirements. Therefore, a UI element describes the other if it is contained within the other.

With this new metric, we produced an algorithm, shown in Algorithm 5, to match a label with an input field on mobile apps. In principle, our matching algorithm is a map function MATCH*(field, state)* $\rightarrow$ *concept*, which receives an input field and a UI state and returns a concept. For our abstraction, we consider as a UI state the set of all UI elements on an app screen.

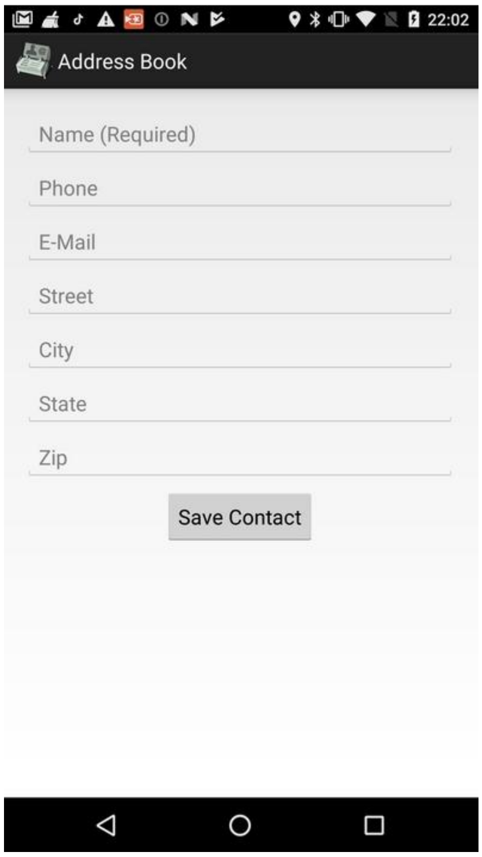


Figure 30: Self-explanatory input fields. Due to limited space, Android apps frequently use the hint text property to describe the input field

Algorithm 5 Matching of an input field to a concept

```
 1: function MATCH(field, state) → concept
 2:     if (field, state) $\notin$ memory then
 3:         if hasText(field) and hasNoun(field) then
 4:             label ← field                                  ▷ Enclosure
 5:         else
 6:             label ← becce(field, state)    ▷ Proxim., Homogen., and Closure
 7:         end if
 8:         result ← concept(best)
 9:         memory ← ((field, state), label)
10:     else
11:         result ← memory(field, state)
12:     end if
13:     return result
14: end function
```

Our algorithm starts by checking if the input field to be matched has not already been processed (line 2). This check is necessary because our *enclosure* metric uses the hint text as a descriptor. Android does not know if an input field has been filled or not; that is, it does not differentiate between the hint text and a typed value on an input field. Moreover, it is no longer possible to determine the original label of an input field once it is filled. We thus create a *memory* with all previously encountered input fields, alongside their matched labels. This allows us to reuse the original label description in case the field gets filled in the future.

If the input field is in the memory, we simply return the previously mapped concept (line 11). Otherwise, if the input field is processed for the first time, we match it to a concept. We first attempt to match it with our *enclosure* metric. If the input field has a text and this text contains a noun (line 3), we define this text as its label (line 4). We consider only labels that contain a noun as candidates descriptors because nouns are used to define objects. Otherwise, we apply Becce's original metrics (line 6) to search for the most relevant label descriptor. Finally, we extract the label's concept, according to Section 6.2.2, and add it to our cache, preventing the input field from being mapped again (line 8-9).

6.2.2 Extracting Concepts

Since we match input fields to both external widgets, as well as hint texts, a label can contain information such as *City* or *Name (required)* as previously shown in Figure 30, or more complex information such as *Enter your username* or *Type a location*. As a consequence, we must pre-process the label to extract a concept from it. Our approach is shown in Algorithm 6.

Algorithm 6 Extracting the concept of a label

```
1:  function CONCEPTS(label) → concept
2:      tagged ← part-of-speech(label)
3:      for candidate in nouns(tagged) do
4:          lemma ← lemmatization(candidate)
5:          value ← link(lemma)
6:          if $|value| = 0$ then
7:              synonyms ← synonyms(lemma)
8:              for synonym in synonyms do
9:                  value ← link(synonym)
10:                 if $|value| > 0$ then
11:                     return value
12:                 end if
13:             end for
14:         else
15:             return value
16:         end if
17:     end for
18:     return $\emptyset$
19: end function
```

We employ natural language processing (NLP) techniques [107] to extract the concept of a label. First, we use *part-of-speech tagging* [109] to identify all nouns of a label (line 2). We then take the first noun as the candidate concept (line 3). We use lemmatization [110] to reduce this noun to its inflectional form and query the available classes and predicates of the knowledge base using this lemma (lines 4-5). If the lemma is found in the knowledge base (line 6), we use it as the label's concept. Otherwise, we search for synonyms in a dictionary (line 7) and systematically query the knowledge base for each synonym, until a result is found in the knowledge base or there are no synonyms left (lines 8-13). If we found a result for any synonym, we return the result as the label's concept (line 11). If we did not, we proceed to the next noun (line 3). If we do not find any result for any of the nouns, we return an empty label (line 18).

6.2.3 Obtaining Input Values

We leverage LINK to obtain semantically aware input values to use during testing once we extracted all input fields and their concepts. LINK consumes our concepts and uses the knowledge base to identify the largest subset of interconnected concepts. By identifying elements which are interconnected, it ensures the semantic coherence of part of the inputs.

Consider our motivating example. LINK can associate the concepts: *author*, *title* and *ISBN* and query for semantically coherent values, such as (Sun Tzu, The Art of War, 9781590302255). It is not, however, able to associate the term *publisher* with them. We overcome this limitation by recursively using LINK, until all concepts which exist in the knowledge base produce input values, either interrelated to other concepts or independently. We summarize our approach to query a knowledge base for input values in Algorithm 7.

Algorithm 7 Querying a knowledge base for candidate input values for a set of concepts

```
1: function QUERY(concepts) → input values
2:     largest-set ← link-associate(concepts)
3:     values ← link(largest-set)
4:     if | (concepts \ largest-set) | > 0 then
5:         values ← QUERY(concepts \ largest-set)        ▷ Recursion
6:     end if
7:     return values
8: end function
```

Our approach starts with a set of concepts to query, and it recursively queries the knowledge base until all concepts have been used. It first uses LINK's concept association feature to obtain the largest set of interrelated concepts (line 2) and invokes LINK to obtain input values for these concepts (line 3). If there are remaining concepts to query, it recursively invokes itself, passing only the remaining concepts (lines 4-6). Finally, it returns the list of queried input values (line 7).

6.2.4 Consuming Input Values

Users expect app functionality to be triggered when interacting with specific types of UI elements [80]. They expect apps to trigger some functionality when they press a button or click on an image. They seldom expect anything but input validation to happen when they enter data on an input field. Moreover, app UIs are designed to guide the user towards specific flows, making the app intuitive. For example, users fill out forms sequentially, with apps guiding them to the next field after entering a value. Under these premises, we intuitively split the UI elements into two categories: input and non-input fields.

To test an app UI, we then first enter values in all input fields for which we successfully queried an input value. We fill the input fields from top to bottom and left to right to emulate the behavior of a user. Once we have filled all possible input fields, we randomly interact with the remaining UI elements to access functionality.

6.3 Evaluation

In this chapter, we presented our approach to obtain real-world input values for testing automatically. This section aims to gather empirical evidence that such inputs can be obtained automatically and lead to better tests. More specifically, we aim to answer the following research questions:

RQ1 (Associating and Extracting Concepts) Can semantic concepts be accurately associated with input fields?

RQ2 (Obtaining Input Values) Can syntactically valid and semantically coherent textual inputs values be extracted from a knowledge base automatically?

RQ3 (Consuming Input Values) Do textual inputs, automatically extracted from a knowledge base, improve test generation?

6.3.1 Experimental Setup

To evaluate our approach, we implemented Saigen (Semantic Aware Input Generator) as a DM-2 [53] plug-in. To identify the UI states and elements on the app, we relied on DM-2's uniqueness measurement, which allows the same UI element to be re-identified between different states. Moreover, for our experiments, we consider as input fields only UI elements of class *android.widget.TextView*, Android's native input field. We used as a knowledge base DBPedia [106]—a structured source of information gathered from Wikipedia—and we used WordNet as a synonym dictionary.

For this evaluation, we could not reuse all apps from the previous evaluations (Chapters 3 to 5), as they do not contain input fields. Therefore, we previewed 120 Android apps on Google Play Store and F-Droid from June to July 2018. We then filtered out those apps with less than 10,000 downloads and remained with 85 apps. We explored these 85 apps using DM-2's random strategy for 500 actions and filtered out those in which the exploration did not reach any native Android text field. We used 500 actions as a limit as previous work [53] showed only a marginal discovery of new functionality after this point. After these filtering steps, our dataset contained 26 apps across different domains, including travel, music, tools, books, games, business, and cars. These apps (henceforth *test set*) and their information are shown in Table 7.

Finally, we executed all experiments on a set of Google Nexus 5X and Google Pixel XL devices, running Android 7.1.2. To prevent device-dependent behavior, all tests for the same app were executed on the same device.

6.3.2 RQ1: Associating and Extracting Concepts

Our first research question aims to measure the accuracy of our label matching and concept extraction approaches, as they have a high impact on our remaining studies. With this goal, we re-executed DM-2's default exploration strategy for 500 actions on all 20 apps from the *test*

Table 7: Selected Applications for the experiment (M for Millions)

Name	**Domain**	**Downloads**
Trip.com	Travel	1M+
Booking.com	Travel	100M+
Agoda	Travel	10M+
Book Catalogue	Books	100,000+
Yelp	Travel	10M+
Kayak	Travel	10M+
Arnab	Tools	100,000+
Youtube Music	Music	50M+
Lonely Planet Guides	Travel	500,000+
TripAdvisor	Travel	100M+
Airbnb	Travel	10M+
Expedia	Travel	10M+
My Books - Library	Books	50,000+
CLZ Games	Games	10,000+
Nader	Tools	50,000+
Rakesh	Tools	10,000+
Careerjet Job Search	Business	1M+
All Job Search	Business	50,000+
AnyCar	Cars	1M+
Jamendo	Music	100,000+
Cirtru	Lifestyle	10,000+
Carlo	Autos	10,000+
arXiv eXplorer	Books	10,000+
Hyderabad Jobs	Business	10,000+
WorkAbroad Interactive	Business	10,000+
Everycar	Lifestyle	10,000+

set while recording (screenshot) all input fields found and their matched label descriptors[22]. We then manually classified each input field found according to the following rules:

- **True Positive (TP)** if the label matches the correct input field;
- **False Positive (FP)** if the label does not match the correct input field, and there is a textual label for this input field on the screen.
- **True Negative (TN)** if the input field was not matched to any label, and there was no textual label matching label on the screen;
- **False Negative (FN)** if the input field was not matched to any label, however, there was a valid textual label for it on the screen.

We applied the rules according to the point of view of a human, accounting for a limitation of our implementation: we do not associate input fields with images. While it is still possible for a human to extract concepts from images, our implementation works only with textual contents. Therefore, if our approach was unable to match an input field because an image instead of a

[22]We ignored fields with the following concepts: username, password, and email, as they are intentionally not available on the knowledge base.

text identified it, we classified this as a true negative. Our reasoning for this choice is that the algorithm did not incorrectly associate the input field with an incorrect label.

We, however, consider as false-negative, situations where our approach could not find a textual label for an input field because it did not contain any noun, such as *Flying to* or *Where to?* While our approach disregards labels without nouns, it is intuitive for a human to associate *Flying to* to the destination of a flight.

Since the evaluation was manual and thus subject to human bias, we performed three independent evaluations for each field-label pair and selected as the final result the one with more votes.

Our findings are shown in Table 8 and summarized in Table 9. Our experiment identified 322 unique input widgets, of which 247 were matched ($\approx 75\%$). Overall, our matching algorithm achieved 87% precision and 94% recall, with 86% accuracy. These input fields represented 1.2% of the total number of unique UI elements found during the tests, which is in line with previous research [111] that demonstrated that less than 3% of the UI elements on apps are input fields.

The per-app breakdown of our results shows that most apps yielded high true-positive and true-negative values. Our manual evaluation of the false positives showed that some apps used noun-less labels, such as *Where to?*, to identify the content of an input field, as shown in Figure 31a. Other false positives happened when hints were used to exemplify inputs, as shown in Figure 31b. Both false-positive examples highlight the limitations of our approach.

SAIGEN matched 75% of the input fields with 87% precision.

6.3.3 RQ2: Obtaining Inputs

Our second research question's goal is to assess the quality of querying a knowledge base for input values. We explored the apps in our test set using DM-2's default exploration strategy for 500 actions with this goal. During the exploration, we recorded the input values obtained for each input field matched from the RQ1.

We then manually assessed the syntactic and semantic quality of the inputs we obtained from the knowledge base. We considered input as syntactically valid if it has passed all app input validation checks (no validation errors triggered).

We considered it as semantically valid if it matches the label associated with the input field. Similar to the previous research question, we analyzed our results from a human perspective and performed three independent analyses to mitigate the inherent bias of manual evaluations. We randomly chose twelve apps from our *test set* for this evaluation due to the significant manual effort involved in analyzing each input value queried from the knowledge base.

Table 10 shows the number of unique labels found on each app, as well as the number of unique labels that were successfully queried. Using DBPedia, we were able to find an input for, on average, 48% of the labels found at least once during testing, with the worst app (Trip.com) finding only 23%. Note that the total number of labels is less or equal to the number of text fields found in the app (Table 8). This happens because the same label can be reused on different input fields.

ht

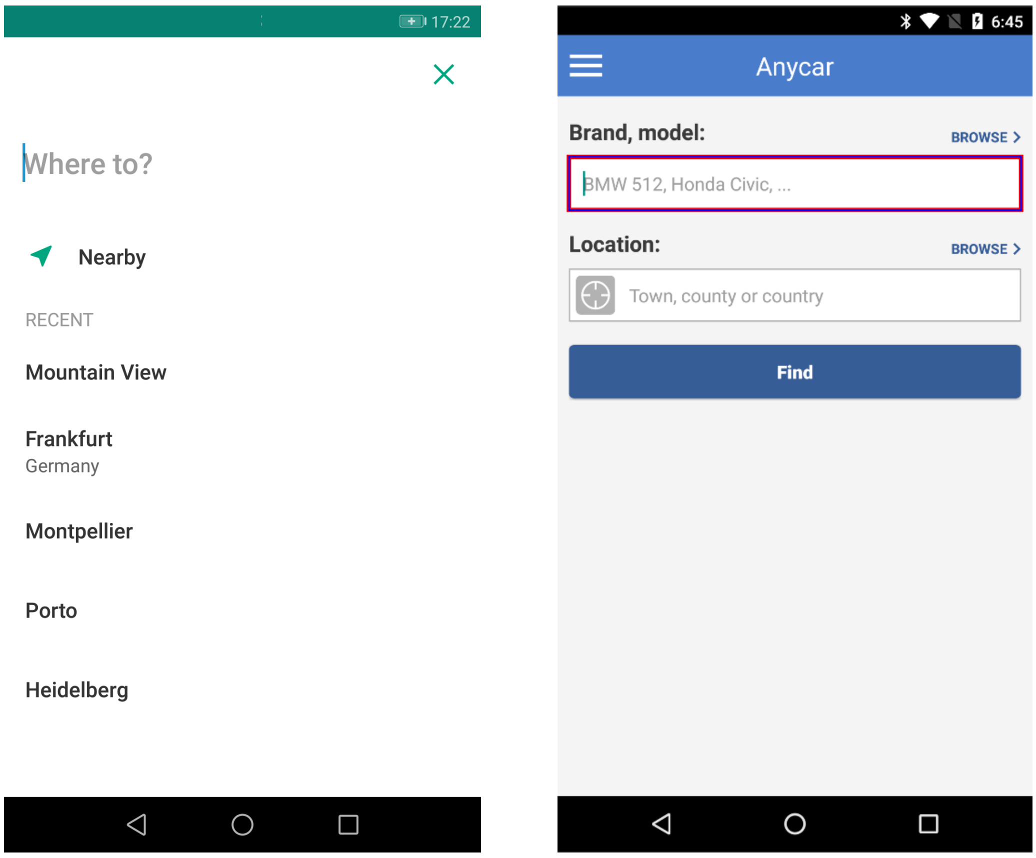
Table 8: Per app breakdown of input field matching

Name	**Fields**	**Matched**	**TP**	**TN**	**FP**	**FN**	**Prec.**	**Recall**	**Specif.**	**Accuracy**
Trip.com	18	15 (83%)	11	3	4	0	73%	100%	43%	78%
Booking.com	14	5 (36%)	5	8	0	1	100 %	89 %	100 %	93 %
Agoda	2	1 (50%)	1	1	0	0	100 %	100 %	100 %	100 %
Book Catalogue	26	24 (92%)	20	1	4	1	83 %	50 %	20 %	81 %
Yelp	15	6 (40%)	5	9	1	0	83 %	100 %	90 %	93 %
Kayak	12	8 (67%)	7	4	1	0	88 %	100 %	80 %	92 %
Arnab	10	9 (90%)	9	1	0	0	100 %	100 %	100 %	100 %
Youtube Music	4	4 (100%)	3	0	1	0	75 %	-	0 %	75 %
Lonely Planet Guides	10	9 (90%	8	0	1	1	89 %	0 %	0 %	80 %
TripAdvisor	12	11 (92%)	11	0	0	1	100 %	0 %	-	92 %
Airbnb	7	5 (71%)	3	2	2	0	60 %	100 %	50 %	71 %
Expedia	27	24 (89%)	23	1	1	2	96 %	33 %	50 %	89 %
My Books - Library	7	7 (100%)	5	0	2	0	71 %	-	0 %	71 %
CLZ Games	15	15 (100%)	14	0	1	0	93 %	-	0 %	93 %
Nader	7	7 (100%)	7	0	0	0	100 %	-	-	100 %
Rakesh	15	8 (53%)	7	7	1	0	88 %	100 %	88 %	93 %
Careerjet Job Search	9	9 (100%)	9	0	0	0	100 %	-	-	100 %
All Job Search	14	12 (86%)	9	1	3	1	75 %	50 %	25 %	71 %
AnyCar	18	16 (89%)	13	2	3	0	81 %	100 %	40 %	83 %
Jamendo	8	7 (88%)	5	1	2	0	71 %	100 %	33 %	75 %
Cirtru	12	8 (66%)	7	4	1	0	88%	100%	80%	92%
Carlo	5	2 (40%)	2	1	0	0	100%	-	-	100%
arXiv eXplorer	3	2 (66%)	2	0		1	0 67%	100%	67%	0%
Hyderabad Jobs	25	14 (56%)	14	7	0	4	100%	78%	100%	84%
WorkAbroad Interactive	16	11 (68%)	10	2	1	1	91%	91%	67%	86%
Everycar	11	8 (72%)	6	1	2	0	86%	100%	67%	89%
Total	**322**	**247 (75%)**	**216**	**56**	**31**	**13**				

Table 9: Unique label descriptor to input fields matching

	Classified as		
Input	**True**	**False**	**Total**
True	TP = 216	FN = 13	229
False	FP = 31	TN = 56	87
Total	247	69	316

Precision = 87%
Recall = 94%
Accuracy = 86%
Specificity = 64%



(a) Label without noun (b) Noun which is not a concept

Figure 31: App functionality which requires syntactically and/or semantically correct values to bypass validations

Regarding the number of queries and the quality of the results, our results are shown in Table 11. We executed 479 queries for the 204 unique input fields matched (Table 8). This number is higher than the overall number of unique input fields because the same fields can be used in different queries. Considering our initial example, the concepts *Author*, *Title*, and *ISBN* are used together on the example screen, as well as alongside the label *Date* (from date published) on a different screen, resulting in two different queries. The knowledge base was able to return input values for 397 out of our 479 queries (≈ 83%), of which we classified 390 values as syntactically correct (≈ 98%) and 375 as semantically coherent (≈ 94%).

83% of Saigen*'s queries were able to find a result in the knowledge base. 98% of the results were syntactically valid, and 94% of them were semantically valid.*

Table 10: Per app breakdown of unique labels and number of labels found on the knowledge base

Name	**Unique**	**Found**	**Ratio**
Trip.com	13	3	23%
Booking.com	3	2	67%
Agoda	1	1	100%
Book Catalogue	24	13	54%
Yelp	4	2	50%
Kayak	4	2	50%
Arnab	8	7	88%
Youtube Music	2	1	50%
Lonely Planet Guides	6	2	33%
TripAdvisor	9	4	44%
Airbnb	7	6	86%
Expedia	20	8	40%
Cirtru	5	3	60%
Carlo	3	1	33%
arXiv eXplorer	3	1	33%
Hyderabad Jobs	8	3	37%
WorkAbroad Interactive	9	4	44%
Everycar	8	3	37%
Total	**137**	**66**	

6.3.4 RQ3: Consuming Input Values

The goal of our final research question is to measure if automatically extracted textual inputs improve Android testing. Moreover, it aims to measure the benefits of orderly interacting with the app UI. In this work, we used code coverage as an indication of the test quality, as previous researches [61] has shown it to be a good predictor. We obtained the code coverage from DM-2's native code instrumentation metrics.

After instrumentation, many apps in our *test set* could no longer be used. This issue arises because many popular apps have safety verifications in place, such as certificate checks, which render the app unusable once instrumented. In the end, 11 out of our 20 apps could be instrumented with DM-2, namely: *My Books Library*, *Rakesh*, *Kayak*, *Arnab*, *Book Catalog*, *Cirtru*, *Carlo*, *arXiv eXplorer*, *Hyderabad Jobs*, *WorkAbroad Interactive*, and *Everycar*.

For this experiment, we explored each app 11 times in each of the configurations described in Table 12, to mitigate noise caused by the seed selection and by app non-determinism, resulting in 20 executions per app. In each test of each app, we executed 500 actions ($\approx$ $30min$), as previous researches [53, 68, 82] showed this to be enough to reach over 95% of the maximum test coverage. We compared the explorations by the number of actions because the overhead of querying DBPedia can be significantly reduced by hosting it locally for testing.

The results of our experiments are summarized in Figure 32. The random exploration with random textual inputs (Scenario 1) achieved an average statement coverage of 38%, with a minimum of 4% and a maximum of 74%. Scenario 2, which replaces random inputs for SAIGEN generated inputs but still retains DM-2's random exploration strategy, achieved an average coverage of 45%, with a minimum of 7% and a maximum of 74%. These results indicate that syntactically valid and semantic coherent inputs are beneficial to the tests, with an average

Table 11: Per app breakdown of syntactically and semantically valid inputs queried

Name	**Queries**	**Found**	**Ratio**	**Syntactically Valid**	**Ratio**	**Semantically Valid**	**Ratio**
Trip.com	23	13	57%	13	100%	13	100%
Booking.com	13	12	92%	12	100%	12	100%
Agoda	6	6	100%	6	100%	6	100%
Book Catalogue	44	33	75%	32	97%	32	97%
Yelp	8	6	75%	6	100%	6	100%
Kayak	19	15	79%	15	100%	15	100%
Arnab	138	137	99%	137	100%	135	99%
Youtube Music	5	4	80%	4	100%	4	100%
Lonely Planet Guides	39	35	90%	35	100%	23	66%
TripAdvisor	20	15	75%	15	100%	14	93%
Airbnb	21	19	90%	15	79%	15	79%
Expedia	24	12	50%	12	100%	12	100%
Cirtru	26	23	88%	23	100%	23	100%
Carlo	10	6	60%	5	83%	5	87%
arXiv eXplorer	5	4	80%	4	100%	4	100%
Hyderabad Jobs	19	15	78%	15	100%	15	100%
WorkAbroad Interactive	39	27	69%	26	96%	26	96%
Everycar	20	15	75%	15	100%	15	100%
Total	**479**	**397**	**83%**	**390**	**98%**	**375**	**94%**

Table 12: Coverage test scenarios

#	**Textual Input generation**	**Widget Selection**
1	Random	Random
2	SAIGEN	Random
3	Random	Sorted
4	SAIGEN	Sorted

increase of 7%.

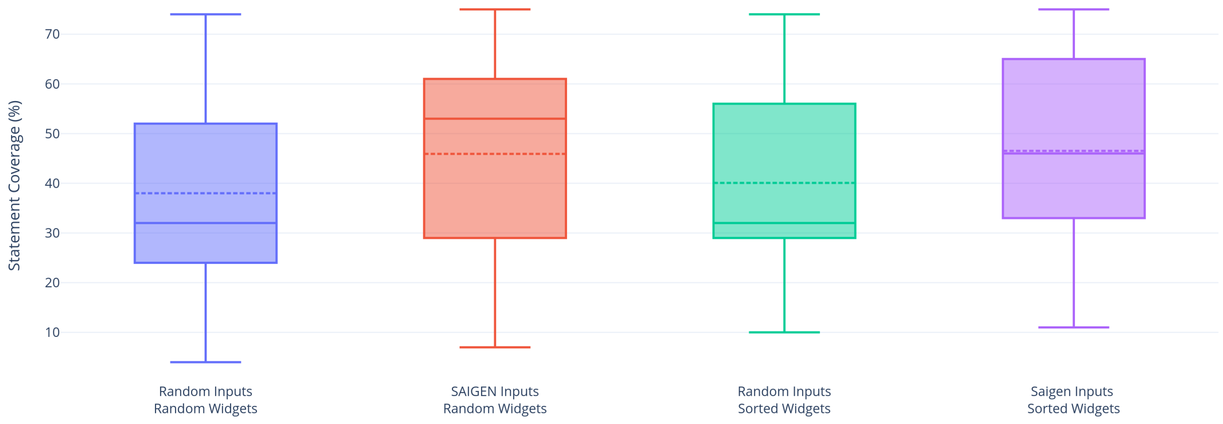


Figure 32: Coverage (%) per experimental scenario

Using randomly generated textual inputs, alongside our sorted exploration order (Scenario 3), achieved an average coverage of 41%, with a minimum of 10% and a maximum of 74%. These numbers are marginal improvements over DM-2's random strategy concerning average coverage (3%) but are a significant increase in the minimum test coverage by 6%. Compared against Scenario 2, random inputs with a sorted widgets interaction achieve, on average, 4% less coverage; however, it still increases the minimum test coverage by 3%. These results indicate that orderly interacting with widgets after filling out the text fields affects the test coverage. However, it also shows that the value entered on the input field is more important than the interaction order.

Finally, in Scenario 4, when combining the sorted exploration order with SAIGEN, we achieve an average statement coverage of 48%, with a minimum of 11% and a maximum of 75%. These results again show the benefits of using valid input values, outperforming all our previous test scenarios. When compared to random input values and interaction order, Scenario 4 achieves, on average, 10% more coverage, with a 7% increase in the minimum coverage. Compared to Scenario 2 (SAIGEN inputs and random interaction order), the minimum coverage obtained is increased by 4%, and the average coverage is increased by 3%.

To ensure the statistical significance of our results, we performed a Friedman Test, which resulted in a p-value < 0.00001, meaning that the results are significant at 1%.

Filling input fields with SAIGEN *generated values before interacting with other widgets improve code coverage by an average of 10%.*

6.3.5 Threats to validity

Regarding external validity, our experiments have demonstrated evidence that the Saigen generated textual input values can significantly improve the test coverage with a set of benchmark apps. However, we cannot ensure that the results generalize to all apps and testing tools. We selected apps from different sizes and categories to mitigate this threat. Additionally, we added Saigen to random test generation approaches; thus, our results are limited by their constraints, such as their inherent inability to perform complex tasks. Saigen, however, can be used alongside any test generation approach to fill input fields automatically.

Regarding construct validity, in the process of extracting a concept from a label, we use a dictionary to identify synonyms for concepts that are not in the knowledge base. We observed that the synonyms acquired from the dictionary are sometimes not flexible enough for our use. Our approach is, however, abstract concerning how to obtain synonyms. Additionally, a notable limitation to our work includes the inability of Link to generate queries when multiple words are used as a query input for the knowledge base. As it stands, the approach requires a single word to be used as a query input for each concept. This can be problematic when the concepts that Saigen extract from the label descriptor contains more than one word per concept or contains multiple concepts. This limitation causes the loss of semantics and may result in the returned values being inaccurate, as explored in [112].

Regarding internal validity, we opted for DM-2's native bytecode instrumentation to acquire statement coverage, being able to test our approach with both open source and commercial apps. Our tests showed that some apps have failsafe mechanisms, such as certificate checks, to prevent app repackaging. A more accurate coverage measurement can be obtained by using the app source code instead of its bytecode and by measuring coverage on native and JavaScript components.

6.4 Limitations

Our experiments showed that using real-world values for input fields leads to improved test coverage. Nevertheless, our implementation and approach do not work for all apps. Our implementation inherits the limitations of DM-2, which we previously discussed in Section 3.7. Additionally, our approach is limited concerning its use of semantics. As we showed in Section 6.3.2, our approach does not consider the semantics of the app under test or those of the whole UI when obtaining input fields. While a human can, for example, associate a label *From* in a travel app with a location, our approach does not. The extraction of semantics on apps and UIs is an ongoing research topic [113, 114, 115].

Another limitation of our approach is image semantics. We currently consider only textual semantics expressed by nouns. However, some apps use icons instead of text for better usability. These apps will not work with our approach. These are the same limitations we encountered and discussed in Section 4.4.

6.5 Related Work

This section summarizes the most relevant works related to this research.

Closest to the approach we presented in this chapter is LINK [14], which we extended. LINK exploits semantic data available on the web to obtain complex values for input fields. In this work, we reused LINK's core concepts, while adapting it to the particularities of Android testing. Another closely related approach is presented in McMinn *et al.* [116], where the authors query a search engine using labels from a UI to obtain input values. Our approach also uses information from the web. While using a search engine leads to more possible results than querying a knowledge base, knowledge bases can be hosted locally, making the overhead of searching an input value negligible. Moreover, since the entries on a knowledge base contain associations, it is possible to query for multiple, semantically correlated values at once, such as requesting a zip code, name of a city and a country to fill a registration form.

This chapter is also loosely related to the existing test generation approaches we previously presented in Section 2.2. Such test generators can benefit from knowing which input values to fill. *Random testing* techniques, such as MONKEY [30], DYNODROID [31], and DM-2 [34], create random sequences chains of events to explore an app. This includes random strings to input on text fields. Our approach can be used alongside such test generators to input meaningful values instead of random strings, enabling them to bypass some input validations.

Model-based testing techniques infer app models using static, dynamic, or hybrid analysis. GUIRIPPER [35] and MOBIGUITTAR [36] create a state machine model of the app while testing it. ORBIT [42] follows a similar approach but uses static analysis to reduce the number of GUI elements to test. SMARTDROID [43] relies only on statically generated activity and function call graphs to identify paths that should be explored. Our approach can be used alongside such techniques. By learning which input values should be entered on an input field, techniques that dynamically extract the model can cover more code coverage in the app. Similarly, techniques that rely on static analysis to build the app model also benefit from such approach, as input validation frequently occurs externally—by checking the result of a web service or database query, for example—and cannot be statically extracted.

Systematic testing approaches employ various algorithms to test applications exhaustively or generate tests that trigger specific behaviors. Tools which use search-based algorithms EVO-DROID [48] and SAPIENZ [49] can use meaningful and random input values to generate a more diverse set of inputs. Approaches that follow a predefined exploration order, such as CRASH-SCOPE [52], can also bypass input validation rules when entering meaningful input values. Finally, approaches that rely on concolic testing or symbolic execution [47, 46] also benefit from inputting meaningful values during testing, as some input validation rules are external to the app and cannot be determined.

6.6 Lessons Learned

This chapter closes the Part II of this thesis, Learning the Language of Apps. More specifically, it presented an approach to address the problem of determining *which input values should be typed* by automatically obtaining real-world inputs from a knowledge base.

It presented how to extract the concepts associated with input fields on Android and how to use them to search a knowledge base for input values, which can be used during testing. Our results showed that even on mobile devices, it is possible to correctly identify over 95% of the labels associated with input fields; and that entering these values on the forms from a user perspective led to an average coverage improvement of 10%.

While our experiments were conducted in DM-2, our approach is not tied to any specific test generator or strategy and can be used alongside other tools. Our approach builds upon previous research for desktop and web applications, adapting it towards Android peculiarities. There is still much room for improvements and future work:

Non-Textual UI Semantics. Our label matching approach attempts to identify a textual element to be paired to an input field. Due to size constraints, developers frequently use non-textual objects, such as images, to describe input fields. Extracting UI semantics from non-textual elements on the GUI can allow for more accurate matching of input fields and labels.

Enhanced Concept Extraction. The concept extraction algorithm uses standard natural language processing techniques, such as part-of-speech tagging and lemmatization. It can benefit from more advanced textual concept extraction approaches that derive a context from the textual content. A brief survey of such techniques is presented in [117].

Enhanced Exploration Strategies. We explored the use of semantically aware inputs alongside a random test generation approach. The same approach can be used with model-based or explorative strategies to trigger more complex app functionality.

Reproducibility: To facilitate replication and extension, all our work is available as open source. The replication package is available at:

`https://github.com/uds-se/droidmate-saigen`

Part III

Modeling the Language of Apps

Chapter 7

Testing With User Interface Grammars

This chapter is taken, directly or with minor modifications, from our submission *Testing With User Interface Grammars* which is currently under review at ACM Transactions on Software Engineering and Methodology (TOSEM) with number TOSEM-2020-0062 [118]. My contribution in this work is as follows: (I) original idea; (II) partial implementation; (III) evaluation.

Graphical user interfaces integrate *graphical* interactions such as mouse clicks with *textual* interactions such as form entries. To produce inputs to explore functionality behind a GUI, one must know multiple languages:

- The language of *graphical inputs* (i.e., sequences of touch events such as clicks and swipes) comes as *finite-state automaton* denoting which interactions lead to which states;
- The languages of *textual inputs* (e.g., string inputs into a form) come as *formally defined languages* (regular expressions, grammars) denoting legal sequences of characters.

In the previous part of this thesis, we presented techniques to *learn* each of these input languages. The problem is that these two formalisms so far are separate. Consider an application with a simple screen where one can enter an email address and password, for example, Figure 33. A finite-state model for this app (Figure 34) would consist of a sequence of states, whose first two would be reached by entering textual input (“`Type`”) and the last one by subsequent clicking (“`Click`”) on a button.

Properly testing the functionality behind the UI modeled in Figure 34, of course, would include testing whether it works for valid and invalid passwords as well as legal and illegal email addresses.[23] However, a finite-state model, as in Figure 34, does not capture the textual languages of *password* and *email*. For *password*, one could provide a regular expression, which in turn could be converted into its own finite-state machine and thus integrated into Figure 34. For *email*, however, there is no such possibility, as the language of email addresses is defined as a context-free language [119] and needs a *grammar* for adequate representation. Nevertheless, if one wanted

[23]This is a real problem. Many websites erroneously do not accept emails with long top-level domains such as *.education*, *.international* and *.industries*.

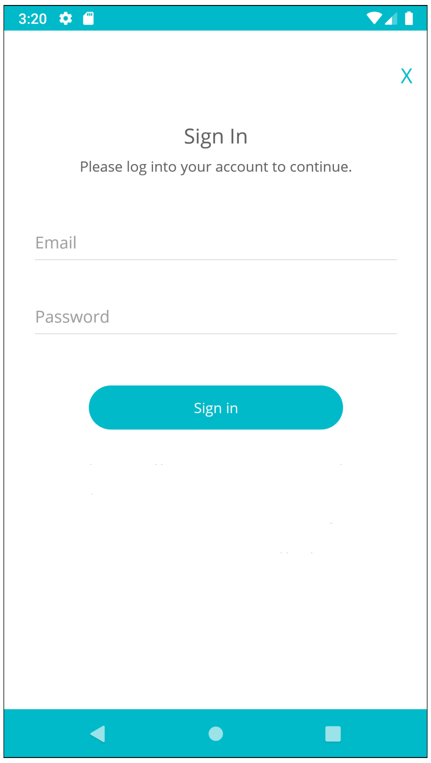


Figure 33: Simple sign in screen

$$\begin{tikzcd}
\text{start} \arrow[r] & s_1 \arrow[loop above, "\text{Type(Password)}"] \arrow[loop below, "\text{Type(Email)}"] \arrow[rr, "\text{Click(Sign In)}"] & & s_2
\end{tikzcd}$$

Figure 34: Finite-state automaton for the simple sign in screen

to test whether any of the fields are vulnerable to overly long inputs, SQL injection, or similar stress tests, a finite-state (sub)model representing these inputs quickly becomes unmaintainable.

In a recent textbook [120], Zeller et al. observed and demonstrated that states, user interactions, and textual inputs could be integrated into a *single formalism* that allows for uniform assessment and production of test cases. The key idea is to use a *context-free grammar* as a single representation of all inputs, in which we *encode* graphical user interactions as sequences of symbols, and we *embed* the finite-state automaton modeling the transitions between states. We can use such grammars as *producers* of interaction sequences that integrate graphical and textual interactions.

Figure 35 shows the excerpt of a grammar integrating the finite-state model from Figure 34 with textual rules for email addresses and passwords. Such grammar models interactions as sequences of individual actions (in our example, *Type()* and *Click()*), which in turn can take textual arguments denoting UI elements and texts to be entered. The action *Type(Password,*

$$
\begin{aligned}
\langle \text{start} \rangle &::= \langle s_1 \rangle && (1)\\
\langle s_1 \rangle &::= \text{Type(Email, } \langle \text{email} \rangle\text{)}\ \langle s_1 \rangle \mid && (2)\\
&\quad\ \text{Type(Password, } \langle \text{pass} \rangle\text{)} \langle s_1 \rangle \mid && (3)\\
&\quad\ \text{Click(Sign In)} \langle s_2 \rangle \mid \varepsilon && (4)\\
\langle s_2 \rangle &::= \varepsilon && (5)\\
\langle \text{pass} \rangle &::= \langle \text{valid-pass} \rangle \mid \langle \text{invalid-pass} \rangle \mid \langle \text{malicious} \rangle && (6)\\
\langle \text{valid-pass} \rangle &::= \langle \text{char} \rangle \langle \text{valid-pass} \rangle \mid \langle \text{number} \rangle \langle \text{valid-pass} \rangle \mid \varepsilon && (7)\\
\langle \text{invalid-pass} \rangle &::= \langle \text{valid-pass} \rangle \langle \text{symbol} \rangle \mid \varepsilon && (8)\\
\langle \text{email} \rangle &::= \langle \text{valid-email} \rangle \mid \langle \text{invalid-email} \rangle && (9)\\
\langle \text{valid-email} \rangle &::= \langle \text{local-part} \rangle @ \langle \text{domain} \rangle && (10)\\
\langle \text{invalid-email} \rangle &::= \langle \text{local-part} \rangle @ \mid @ \langle \text{domain} \rangle && (11)\\
\langle \text{local-part} \rangle &::= \langle \text{dot-atom} \rangle / \langle \text{quoted-string} \rangle / \langle \text{obs-local-part} \rangle && (12)\\
\langle \text{malicious} \rangle &::= \langle \text{valid-pass} \rangle \text{" OR “1” = “1} && (13)\\
&\quad \ldots && (14)
\end{aligned}
$$

Figure 35: Excerpt of grammar modeling UI transitions and textual production rules

⟨pass⟩), for instance, means to type a text into the *password* field—a text whose structure is given by the grammar rule for ⟨pass⟩.

Such an integrated representation for producing UI interactions has several advantages. It allows users to control *what* to test and *how* to test; and to uniformly apply concepts of grammar-based testing across both GUI interaction and textual input. For example:

1. Testers can edit the production rules ⟨valid-pass⟩ (lines 6 and 7) and ⟨valid-email⟩ (lines 9 and 10) to test the app with valid emails and passwords—creating a sequence such as *Type(Email, “e@mail.com”), and Type(Password, “abc123”), Click(Sign In).*

2. Testers can also extend the production rule ⟨malicious⟩ (lines 6 and 13) to test if the app behaves correctly with intentionally malicious passwords, such as *abd" OR "1" = "1*, or use the production rule ⟨invalid-email⟩ (lines 9 and 11) to check if the app rejects invalid emails, such as *invalid@*.

3. Testers profit from other features of grammar testing, such as assigning *probabilities* to individual rules, thus ensuring during production that specific sequences are tested more thoroughly than others [121]. Such probabilities are common features of grammar testing; no specific adaptation is required.

4. Another feature of grammar testing that users profit from is: producers can ensure *coverage* of all grammar rules. In our model, such a producer would cover not only all states and transitions, but also all variants in the textual inputs (say, ⟨valid-email⟩ and ⟨invalid-email⟩), and combinations thereof. Again, achieving such coverage is already a feature of grammar testing tools [122]; no specialization for UI testing is required.

It is worth noting that these benefits do not come from some specific tool or its implementation, but rather *from the model itself.* Embedding a finite-state automaton into a grammar preserves all its original features (it can also be extracted as such again), just like the grammar easily captures all lexical and syntactic properties of textual inputs. But can we also *extract* such complex models from existing applications? And can we successfully *test* using such models?

In this work, we build on the concept of UI grammars to explore how to *mine* UI grammars and *test* with UI grammars on real-world applications, using the Android mobile platform as an example domain. After introducing how to apply UI grammars on the Android user interface (Section 7.1), we make the following contributions:

Mining UI Grammars. Given an app with a graphical user interface, we automatically mine UI grammars from system tests, which can be either manually written or automatically generated (Section 7.2). In our experiments, we tested 46 open-source Android apps automatically, and we successfully mined UI grammars from all test cases. Moreover, we measured the accuracy of our grammar by using it to produce new tests that execute specific interactions from the original tests. 82% of these test cases successfully reach their target.

Generating Tests from UI Grammars. We show how established grammar fuzzing techniques can be applied to the mined UI grammars (Section 7.3). Our experiments show that grammar-generated inputs cover the UI and code more efficiently than the test case from which it was mined. Compared to the original test case, inputs generated by a UI grammar need only 23% of the events to reach 80% of the UI events and 83% of the lines of code.

Associating Grammar Elements with Covered Code. We associate the lines of code reached by each UI element with each grammar production, allowing grammar fuzzing techniques to not only target previously reached UI events but to target the previously reached lines of code (Section 7.4). Moreover, we exploit this association between UI grammar and reached lines of code to guide test generation towards specific functionality—for instance, those impacted by a code change during regression testing—producing multiple inputs that efficiently trigger specific lines of code. During our experiments, 70.5% of the inputs generated by a mined code grammar can reach their target line of code. Moreover, they need only 11% of the events to reach 79% of the lines of code, when compared to the original test case.

After discussing limitations (Section 7.6) as well as related work (Section 7.7), we present our conclusions (Section 7.8).

7.1 User Interface Grammars

The graphical user interface (UI) of an Android app is composed of graphical and structuring elements, commonly referred to as widgets, which are grouped in activities or fragments (states), and displayed. A user interacts with an app through UI-events, such as a click or a swipe, which are handled by the UI state or one of its widgets.

Interacting with an Android app can thus be modeled as producing and applying valid sequences of events. On an abstract level, widgets and UI states describe the app's *input language*; that is, they determine the set of valid sequences of input events. Traditionally, UIs are modeled as finite-state models (FSM), where nodes represent UI states, and transitions represent actions which may lead the app to other UI state [35, 37, 44] A FSM suffices to model all types of UI interactions, be they widget clicks, keyboard keys, gestures, or button presses. However, it cannot model the input format of textual fields on the UI, as a regular language cannot always represent them (e.g., an *email*).

In formal language theory, grammars are well-known formalisms to specify input languages. On the Chomsky hierarchy for grammars [12], context-free grammars (CFG) are the formalism of choice to model languages that cannot be encoded as FSM, but that do not require context-sensitivity. A grammar consists of a start symbol, ⟨start⟩ as a convention, and a set of production rules which indicate how to expand symbols [123]. Each grammar symbol can be a non-expandable *terminal symbol;* or a *nonterminal symbol*, which can be expanded into other symbols. We represent nonterminal symbols with their names between ⟨⟩. In a CFG, the left-hand side of a production rule always contains a single nonterminal symbol.

Widgets and UI states describe the input language of the app; that is, the sequences of interactions that can be performed. We thus encode such a language using a CFG that models UI actions as terminal symbols and UI states and transitions as nonterminals, which we name *UI grammar*. On a single state, a UI grammar must support different *interaction types,* targeted or not to a specific UI element, as well as handle *nondeterminism,* because the same action may lead the app to different states. Finally, an input produced by a UI grammar must encode sufficient information to produce a unique app interaction.

To illustrate the usefulness of a UI grammar, consider an app represented by the state machine from Figure 36. This app can start in the *auth* or *main* state, depending on how they are launched. On the *auth* state, the user must enter a valid four-digit PIN, "1234" for example, to continue. If the user enters an invalid PIN, such as "abcd", the app remains in the *auth* state. Moreover, in the *auth* state, the user can press the device's back button or press a *clear* button on the app to clear the typed PIN. In the *main* state, the user can interact with the app by clicking on *menu* or *clear* buttons, as well as by *swiping left.* By *pressing the device's back button*, the app continues in the *main* state or returns to *auth*, depending on much time has passed since the last authentication. Finally, the user can stop using the app at any point in time.

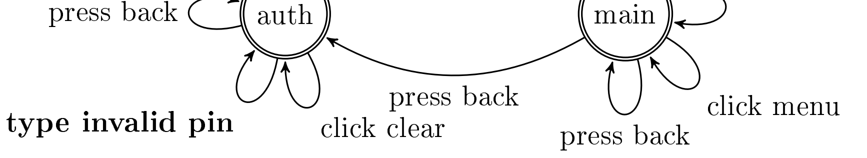


Figure 36: Example app

Note that while this finite-state model is clean and easy to read, it does not specify constraints on textual inputs. In our example, for instance, a valid PIN must be a four-digit number. To express this constraint, one must either supply a separate definition in a different notation, such as a regular expression (say, `[0-9][0-9][0-9][0-9]`), or, to stay within the model, include another finite state diagram such as the one illustrated in Figure 37. If one is to model invalid PINs, the regular expressions and models to be attached become even more complicated.

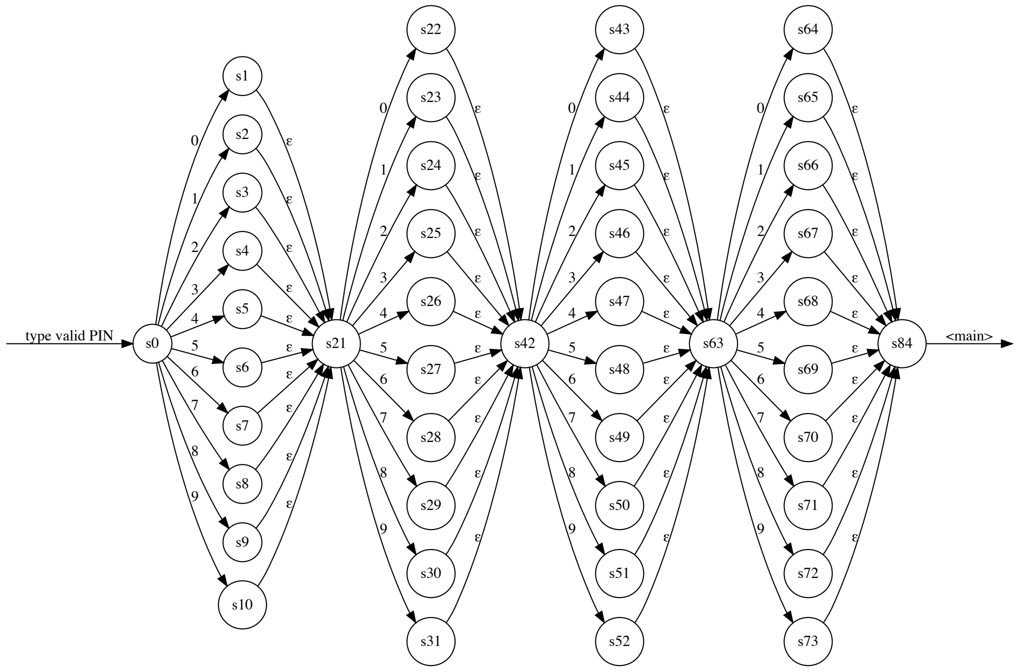


Figure 37: A finite state model to specify that a valid PIN must have 4 digits

Now let us represent this same state model as UI grammar in Figure 38. To this end, we *embed* the finite state model into the grammar, where each FSM state becomes a nonterminal symbol. Each state's transition becomes a possible expansion of the symbol. Transitions become nonterminal symbols, too, expanding to the possible states reached.

- The ⟨start⟩ production rule can lead to either the *auth* or *main* state (Line 1).
- In the ⟨auth⟩ state (Lines 2–6), it is possible to enter a valid PIN and continue to the *main* state (Line 2) or enter an invalid PIN, click on the clear button or press back and continue in the *auth* state (Lines 3–5).
- In the *main* state (⟨main⟩), it is possible to click clear, menu or to swipe left and stay in the *main state* (Lines 7–9). It is also possible to press the back button on the device and go to the *auth* or *main* states (Line 10).
- Finally, it is also possible to stop using the app altogether at any moment (Lines 6 and 11).

The grammar goes beyond a simple embedding, though. In contrast to the FSM, the input specification of a valid 4-digit PIN can be modeled within the grammar (lines 13–14), without the use of an additional model or significant complexity increase. If a user interface accepts more complex inputs, such as an e-mail address, a domain name, or a SQL query, the languages for each of these inputs can be modeled within the grammar—and be used for test generation.

$$
\begin{aligned}
\langle\text{start}\rangle &::= \langle\text{auth}\rangle \mid \langle\text{main}\rangle && (1)\\
\langle\text{auth}\rangle &::= \text{Type(PIN, }\langle\text{valid-pin}\rangle\text{)}\ \langle\text{main}\rangle \mid && (2)\\
&\quad\ \ \text{Type(PIN, }\langle\text{invalid-pin}\rangle\text{)}\ \langle\text{auth}\rangle \mid && (3)\\
&\quad\ \ \text{Click(clear)}\ \langle\text{auth}\rangle \mid && (4)\\
&\quad\ \ \text{Back(auth)}\ \langle\text{auth}\rangle \mid && (5)\\
&\quad\ \ \varepsilon && (6)\\
\langle\text{main}\rangle &::= \text{Click(clear)}\ \langle\text{main}\rangle \mid && (7)\\
&\quad\ \ \text{Click(menu)}\ \langle\text{main}\rangle \mid && (8)\\
&\quad\ \ \text{Swipe(main, left)}\ \langle\text{main}\rangle \mid && (9)\\
&\quad\ \ \text{Back(main)}\ \langle\text{Back(main)}\rangle \mid && (10)\\
&\quad\ \ \varepsilon && (11)\\
\langle\text{Back(main)}\rangle &::= \langle\text{auth}\rangle \mid \langle\text{main}\rangle && (12)\\
\langle\text{valid-pin}\rangle &::= \langle\text{digit}\rangle\langle\text{digit}\rangle\langle\text{digit}\rangle\langle\text{digit}\rangle && (13)\\
\langle\text{digit}\rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 && (14)\\
&\ldots && (15)
\end{aligned}
$$

Figure 38: Equivalent grammar for the example app

7.1.1 Actions

Let us now go further into how to model UI interactions, notably on the Android platform.

To interact with an Android app, one can interact with a specific screen widget by, for example, clicking or entering text into it. One can also perform actions that do not target a specific app widget, such as swiping on the screen or pressing the back button in the navigation bar. It is also possible to stop using the app at any moment or to close it without interacting with the app itself by pressing the home button[24].

In our UI grammar, we model UI actions as *terminal symbols.* More specifically, we encode UI actions as *widget actions*, which target a specific UI element, *state actions*, which targets a UI state and *empty actions*, which allows the test to stop at any moment, without any further interaction.

7.1.1.1 Widget Actions

Widget actions represent actions that target a specific UI element on the screen. They are used to model actions such as clicking, long clicking, or entering text into a specific UI element.

We model widget actions as terminal symbols in the grammar, and we encode them as:

$$action(widget, payload)$$

where *action* is an action type, such as Click, LongClick or Type; *widget* is the UI element which should handle the event; *payload* the action payload, such as a text to be typed or a nonterminal

[24]While apps can also handle system events, such as message/call received, in this work, we model exclusively transitions which happen through UI interactions. However, our model could also be used to model system events by extending the definition of state actions.

grammar symbol to produce input values.

By not including a *state*, this encoding allows the same widget to be re-identified in different states. It supports multiple actions for the same widget or distinct payloads for the same action in the same state.

When used in a UI grammar, a *widget action is always followed by a state or transition nonterminal.* The actions *Type(PIN, ⟨valid-pin⟩)*, *Type(PIN, ⟨invalid-pin⟩)*, *Click(clear)* and *Click(menu)* from our example grammar (Figure 38) are examples of widget actions.

7.1.1.2 State Actions

State actions represent actions that are not directed to specific widgets but instead target a specific UI state. They are used to model inputs such as pressing the back button on the navigation bar, swiping on parts of the screen, or performing other gestures.

We model state actions as terminal symbols in the grammar and we encode them as:

$$action(state, payload)$$

where *action* is an action type, such as Click, LongClick or Type; *state* is the UI state which should handle the event; *payload* the action payload, such as the direction—or start and end of coordinates—for a swipe action, or sequences of coordinates for other gestural events.

The encoding is similar to that of a widget action; however, instead of a widget, it contains a state. This encoding provides similar benefits as the one used for widget actions; that is: we can re-identify actions between different states and express, in the same state, distinct action types or different actions of the same type with distinct payloads.

When used in a UI grammar, a *state action is always followed by a state or transition nonterminal.* In our example grammar from Figure 38, the *Back(auth)*, *Back(main)*, and *Swipe(main, left)* are state actions.

7.1.1.3 Empty Action

The terminal symbol ε denotes these actions. They are used in all production rules in the UI grammar to model that a user may close or stop using the app at any moment, without interacting with any widget of UI state.

In the example grammar from Figure 38, the states ⟨*auth*⟩ (Line 6) and ⟨*main*⟩ (line 11) can be expanded into the empty action.

7.1.2 States

In an app, the UI state of an app determines which UI elements exist and which actions are valid. In our UI grammar, a UI state is modeled by a nonterminal symbol, whose expansions determine which actions can be produced and which other states are reached.

To uniquely identify states, we use identifiers generated from the UI elements on the screen or extracted from the UI semantics. As any nonterminal symbol in a grammar, a state must be defined, i.e., used on the left-hand side of one production rule, once and must be used at least once on the right-hand side of some production rule.

7.1.2.1 Definition

A state nonterminal defines which actions and transitions can be performed on the state. Thus, in a UI grammar, a state nonterminal definition can contain actions, followed by transitions or states on its right-hand side. In addition to any number of actions, *all state nonterminals must be able to produce an empty action*, indicating that the user can stop interacting with the app at any moment.

Our grammar from Figure 38, for example, defines two state nonterminals: ⟨auth⟩ and ⟨main⟩.

7.1.2.2 Use

On the right-hand side of a production rule, a state symbol represents which UI states a transition can reach. In our model, a UI state can only be reached either by starting the app or through a UI transition. Therefore, a state nonterminal can only be used on the right-hand side of a production rule if the left-hand side of the same rule is a transition symbol.

Our grammar from Figure 38 uses the state nonterminals ⟨auth⟩ and ⟨main⟩ on the right-hand side of the transition production rules in lines 12.

7.1.3 Transitions

In an app, each action may transition the app into a different UI state, and the same action may transition the app into different states depending on a wide variety of constraints. On a log-in screen, such as the one in our first example, clicking a log-in button may lead the app into an error state if the username or password is incorrect. It may lead the app into a different error state if the device has no Internet connection. Even when the username and password are correct, after clicking the button, the app may, sometimes, advertise a product for monetization, while on other times leading directly to its logged-in screen.

In our UI grammar, such behaviors are modeled by a transition nonterminal. As any nonterminal symbol in a grammar, a state must be defined once, that is used on the left-hand side of one production rule, and must be used at least once on the right-hand side of some production rule.

We encode a transition nonterminal according to the action which triggered it as:

$$\langle \text{action(state, widget, payload)} \rangle$$

where *action* is an action type, such as Click, LongClick or Back; *widget* is the UI element from the preceding action, if any; *state* is the UI state from the preceding action; *payload* the payload of the preceding action, if any.

7.1.3.1 Definition

A transition nonterminal defines which states can be reached after an action. Thus, in a UI grammar, a transition nonterminal definition can contain state nonterminals on its right-hand side.

Our grammar from Figure 38, for example, defines the transition nonterminal ⟨Back(main)⟩, which can lead to both ⟨auth⟩ and ⟨main⟩ states.

While in our example, we used a transition nonterminal only when the action could lead the app to more than one state. Nevertheless, one could also expand state nonterminals used on the right-hand side of a production rule into transition nonterminals. For example, one could rewrite the ⟨auth⟩ nonterminal definition to use transition nonterminals, with the result shown in Figure 39.

$$
\begin{aligned}
\langle\text{auth}\rangle ::= {} & \text{Type(PIN, }\langle\text{valid-pin}\rangle\text{)}\ \langle\text{Type(auth, PIN, valid-pin)}\rangle \mid && (1)\\
& \text{Type(PIN, }\langle\text{invalid-pin}\rangle\text{)}\ \langle\text{Type(auth, PIN, invalid-pin)}\rangle \mid && (2)\\
& \text{Click(clear)}\ \langle\text{Click(auth, clear)}\rangle \mid && (3)\\
& \text{Back(auth)}\ \langle\text{Back(auth)}\rangle \mid && (4)\\
& \varepsilon && (5)\\
\langle\text{Type(auth, PIN, valid-pin) }\rangle ::= {} & \langle\text{main}\rangle && (6)\\
\langle\text{Type(auth, PIN, invalid-pin) }\rangle ::= {} & \langle\text{auth}\rangle && (7)\\
\langle\text{Back(auth)}\rangle ::= {} & \langle\text{auth}\rangle && (8)\\
\langle\text{Click(auth, clear)}\rangle ::= {} & \langle\text{auth}\rangle && (9)
\end{aligned}
$$

Figure 39: State ⟨auth⟩ using transition nonterminals

7.1.3.2 Use

When used on the right-hand side of a production rule, a transition symbol defines an action's effect. Since only widget or state actions cause UI transitions, a transition symbol must be preceded by an action symbol when used on the right-hand side of a production rule.

7.1.3.3 Grammar Entry point

A grammar must have an entry point, commonly expressed by the nonterminal ⟨start⟩ An app, however, may start in different UI states, according to external conditions. We support such behavior by defining that the ⟨start⟩ symbol of a UI grammar is always a transition symbol, which points to all states which the app can display when launched. In our example from Figure 38 the grammar start symbol (line 1) can transition the app to both ⟨auth⟩ and ⟨main⟩.

7.2 Mining User Interface Grammars From Test Cases

UI grammars describe the GUI and transitions of an app; however, writing a grammar from scratch is a laborious and error-prone task. When test cases are available, be they manually written or automatically generated, UI Grammars can be extracted automatically, as long as the test case can uniquely identify widgets and UI states.

We abstract a test case as a sequence of events. In this abstraction, we assume that an event contains a *source state*, the state the app was before the action; an *action type*, such as Click, Swipe, or Type; an *action payload* with additional data needed by the action such coordinates for swipes or the text to type; a *target widget*, if the action targeted a specific widget and a *reached state*, that is, the state the app reached after the action.

Based on this abstraction, we use the mining algorithm described in Algorithm 8 to extract a UI grammar from such a sequence of events.

Algorithm 8 Mining a UI Grammar from an existing test case

```
 1: function EXTRACT UI GRAMMAR(trace) returns grammar
 2:     for entry in trace do
 3:         (src, action, payload, target, dst) ← entry
 4:         if isLaunch(action) then
 5:             grammar ← ⟨start⟩ ::= ⟨dst⟩
 6:         else
 7:             action ← GET-TERMINAL(entry))
 8:             transition ← ⟨action(src, target, payload)⟩
 9:             grammar← ⟨src⟩ ::= action transition
10:             grammar ← ⟨src⟩ ::= $\varepsilon$
11:             grammar ← transition ::= GET-STATE(⟨dst⟩)
12:         end if
13:     end for
14: end function
15:
16: function GET-TERMINAL(entry) returns terminal
17:     (src, action, payload, target, dst) ← entry
18:     if hasTarget(entry) then
19:         terminal ← action(target, payload)
20:     else
21:         terminal ← action(src, payload)
22:     end if
23: end function
```

We model our mining algorithm as a mapping function which receives a test log (trace) as input and produces a UI grammar. It starts by decoding each entry in the trace into its components, the source (*src*) and destination (*dst*) UI states, the conducted *action* that led from *src* to *dst* and, if the action was targeted towards a UI widget, the *target* widget and its corresponding *payload* (Lines 2-3).

If the action type is *launch*, that is, start the app from the Android launcher, it appends the state resulting from the launch (⟨dst⟩) to the production rule ⟨start⟩ (Lines 4-5). If not, the algorithm decides if the action is a state or widget action by checking if the current trace contains a *target* widget (Line 18) and encodes the action terminal (Lines 7;16-23) and transition nonterminal symbols accordingly (Lines 8).

The algorithm then adds the production rules ⟨src⟩ ::= *action transition* and ⟨src⟩ ::= ε to the grammar (Lines 9-10). Where the first production rule adds a new possible action to the state production rule ⟨src⟩, and the second ensures that an input generated from this grammar can terminate in this state. For simplicity, we assume in our algorithm that a production rule is created if none exists with the same left-hand side symbol; otherwise, the right-hand side of the new production rule is appended to the right-hand side of the existing production rule as an alternative if the same alternative does not already exist.

Finally, the algorithm ends by creating the transition production (Line 11), which points to the next state or to the empty action (ε) if the current entry is the last one on the trace or if the next entry is a launch action.

Note that this algorithm does not extract grammar for input fields, instead entering the original value used on the test case. We opted for this solution as the mining algorithm does not handle app semantics; that is, what would be the correct or incorrect input values? Moreover, without more executions of the test case, it is not possible to know which transitions would occur when typing a different input value. While techniques to extract grammars from test executions exist in the literature [124], they require several test runs, making them unsuitable for real-world apps. Moreover, they rely on a parsing phase to validate the input, which does not apply to mobile apps.

7.3 Guiding Tests Towards User Interface Coverage

Specifying inputs via grammar allows apps to be explored systematically and efficiently. Expansions of a context-free grammar can be efficiently modeled and controlled through a representation known as derivation trees. Similar to other computational tree structures, a derivation tree consists of nodes, which point to other nodes (children). The only node without a parent is denoted root, and the nodes with no children are denoted leaves.

The process of expanding a grammar derivation tree can be abstracted as follows: start from the root node and traverses the tree searching for a nonterminal symbol that is not yet expanded. Choose an expansion for this symbol from the grammar and add it as a new child of the unexpanded node. Repeat this process until there is no nonterminal symbol left to expand. By choosing among different expansions for the same symbol, a grammar can produce different sequences of symbols, which on our UI grammars represent different sequences of events to trigger on the app.

When randomly expanded, grammar production rules are efficient generators of test inputs; however, such inputs may cover the same actions and transitions multiple times. A more efficient approach is to guide the grammar expansion towards specific targets, such as features or unexplored actions.

We employ a grammar-expansion technique based on the derivation trees by exploiting the algorithms presented in the fuzzing book [120]. We updated their textbook algorithms to guide the grammar expansion towards the terminal symbols in the grammar, instead of the nonterminals. With this change, we guide the input generation algorithms to produce values which cover specific UI actions.

We guide the input generation using an iterative deepening depth-first strategy to select the best expansion for each non-expanded node. With this approach, we first cover all children up to a given depth and choose the child, which yields the highest number of newly covered terminals. If no children in depth can produce new terminals, we proceed to the next depth. If two or more children yield the same maximum terminal coverage on a specific depth, we randomly select one to expand. We repeat this process for each non-expanded symbol in the derivation tree until all symbols have been expanded.

Just as the original test case may restart the app to explore new paths, we cannot ensure that the input-generator process will create a single input that covers all terminal symbols in the grammar. Thus, we produce as many inputs as needed to cover all terminal symbols. After producing each input, we reset the derivation tree back to its root node, while retaining the list of reached terminals, and expand it again. We repeat this process until all terminal symbols are reached.

We then convert each derivation tree into a test case. Terminal symbols represent actions on a UI grammar and can be translated into a sequence of events to be sent to the app. To convert a derivation tree into a test case, we concatenate the root node with *all terminals* from the tree following a depth-first search strategy. The result of this approach are inputs such as: ⟨start⟩ → *Type(PIN, 'abcd')* → *Click(clear)* → *Back(auth)* → *Type(PIN, '1234')* → *Click(menu)* → *Back(main)* → *Swipe(main, left)*. Which suffices to cover all terminal symbols from the example grammar in Figure 38.

By guiding the input generation towards terminal symbols, our inputs avoid repeating the same widget action on different states, such as *Click(clear)* in both ⟨auth⟩ and ⟨main⟩ states.

One may not want to generate inputs that cover all actions in the grammar but instead target specific actions or states. While it is possible to adapt the algorithm to expand the derivation tree to target specific goals, a more general solution is to translate the grammar, remove all non-relevant terminals, and use the original expansion algorithm.

We illustrate such a translation in Figure 40. The state production rule ⟨main⟩ from Figure 40a can be expanded into four actions. Assuming, for example, that we are only interested in the actions ⟨Click(main, menu)⟩ and ⟨Back(main)⟩. We can rewrite this production rule without the remaining actions, while retaining all possible transitions, as illustrated in Figure 40b, and use it as a regular UI grammar to produce test cases.

$$
\begin{aligned}
\langle\text{main}\rangle ::= {} & \text{Click(clear)}\ \langle\text{Click(main, clear)}\rangle \mid \\
& \text{Click(menu)}\ \langle\text{Click(main, menu)}\rangle \mid \\
& \text{Back(main)}\ \langle\text{Back(main)}\rangle \mid \\
& \text{Swipe(main, left)}\ \langle\text{Swipe(main, left)}\rangle \mid \\
& \varepsilon
\end{aligned}
$$

(a) *Production rule*

$$
\begin{aligned}
\langle\text{main}\rangle ::= {} & \langle\text{Click(main, clear)}\rangle \mid \\
& \textbf{Click(menu)}\ \langle\text{Click(main, menu)}\rangle \mid \\
& \textbf{Back(main)}\ \langle\text{Back(main)}\rangle \mid \\
& \langle\text{Swipe(main, left)}\rangle \mid \\
& \varepsilon
\end{aligned}
$$

(b) *Rewritten production rule*

Figure 40: Example on how to translate a production rule in a UI grammar to target specific actions (e.g. Click(menu) and Back(main) on the main state

$$\begin{aligned}\langle\text{main}\rangle ::= {} & \text{Click(clear)}\ \langle\text{Click(main, clear)}\rangle \mid \\ & \text{Click(menu)}\ \langle\text{Click(main, menu)}\rangle \mid \\ & \text{Back(main)}\ \langle\text{Back(main)}\rangle \mid \\ & \text{Swipe(main, left)}\ \langle\text{Swipe(main, left)}\rangle \mid \\ & \varepsilon\end{aligned}$$

(a) *UI Grammar*

$$\begin{aligned}\langle\text{main}\rangle ::= {} & \text{loc}_a \ldots \text{loc}_{a'}\ \langle\text{Click(main, clear)}\rangle \mid \\ & \text{loc}_b \ldots \text{loc}_{b'}\ \langle\text{Click(main, menu)}\rangle \mid \\ & \text{loc}_c \ldots \text{loc}_{c'}\ \langle\text{Back(main)}\rangle \mid \\ & \text{loc}_d \ldots \text{loc}_{d'}\ \langle\text{Swipe(main, left)}\rangle \mid \\ & \varepsilon\end{aligned}$$

(b) *Equivalent Code Grammar*

Figure 41: Associating UI grammars with lines of code

7.4 Guiding Tests Towards Code Coverage

We showed how to use inputs produced via a UI grammar to guide test generation towards specific UI elements, as well as to effectively re-trigger all actions from the grammar. Nevertheless, multiple actions can trigger the same underlying lines of code.

Thus, we can assume that a more efficient approach to testing the app code is to guide the test generation towards uncovered code elements, not actions. This, however, requires our UI grammar to be associated not only with UI actions but also with code locations. That is, it requires the underlying app code to be associated with the input elements which trigger it. Therefore, to avoid using a secondary model, we propose an extension of our UI grammars, called *code grammars*.

Code grammars are similar to UI grammars; however, their terminal symbols no longer encode UI actions, such as click and swipes, but instead encode the lines of code reached by that action. At a syntactical level, the only difference between code and UI grammars is the encoding of its terminal symbols, which change from UI actions to lines of code. At a semantic level, state production rules on a code grammar model all code locations that can be reached from a state, while the semantics of the transition production rules remain unchanged.

Figure 41 shows the difference between a UI and a code grammar. Figure 41a shows a state production from a UI grammar containing, besides ε, four possible actions and transitions. As a code grammar (Figure 41b), the production rule no longer has four actions. Instead, each terminal now represents the lines of code (loc_x) triggered when the action is performed.

7.4.1 Mining Code Grammars

It is significantly harder to write code grammars than UI grammar. While apps may support hundreds of different UI actions, they may have hundreds of thousands of lines of code. Ideally, code grammars should be automatically generated from the app source code or its execution. We address this challenge by mining code grammars from test cases.

In Section 7.2, we presented an algorithm to mine UI grammars from existing test cases automatically. We abstracted test cases as sequences of events containing: source state, action type, action payload, target widget, and reached the state. To mine code grammars, we extend this abstraction and assume that the lines of code reached by each action are also available in the event sequence. This information can be obtained, for example, by monitoring the app statements reached during testing (coverage) and associating this information with each action.

With this extension, we adapt Algorithm 8 to mine code grammars by replacing the GET-TERMINAL function (Lines 16-23 in the original algorithm) with the one shown in Algorithm 9. This new algorithm decodes an entry (Line 2) and sets the list of statements (Line 6) as the terminal symbol or ε when this action did not trigger any statement (Line 4).

Algorithm 9 Mining a code Grammar from an existing test case

```
1: function GET-TERMINAL(entry) returns terminal
2:     (src, action, payload, target, dst, stmts) ← entry
3:     if stms is empty then
4:         terminal ← $\varepsilon$
5:     else
6:         terminal ← stmts
7:     end if
8: end function
```

7.4.2 Guiding Tests Towards Code Locations

We previously showed how UI grammars could be used to guide testing towards specific functionality. While manually written tests typically exist to cover critical functionality, such tests may perform many actions and spawn for many minutes.

On a regular development cycle, small changes to the app code are made continuously, and it is expensive to continuously determine which of the existing tests trigger the modified code segment and execute them. Moreover, such tests may trigger the modified code location in a single context. Ideally, one wants to create several and quick tests to trigger specific code segments through multiple contexts. One may then, less frequently, execute more extensive tests for additional guarantees.

Code grammars can be used to generate multiple small inputs to trigger specific code locations with minimal effort. We use a code grammar to produce inputs following the same approach we used on UI grammars (Section 7.3), that is, by expanding a derivation tree through an iterative deepening depth-first strategy. Similarly to inputs produced by a UI grammar, it may be necessary to produce multiple inputs to cover all terminals in the code grammar. Thus, we produce as many inputs as needed to cover all terminal symbols, and then convert produced input into a test case.

On a UI grammar, terminal symbols represent actions and can be translated into a sequence of events to be sent to the app. However, on a code grammar terminal symbols represent lines of code, and cannot be translated into app events. Therefore, to convert a derivation tree into a test case, we expand the grammar to use *transition nonterminals* for all transitions and then concatenate the root node with *all transition nonterminals* from the tree following

$$\begin{aligned}\langle\text{main}\rangle ::= {} & \text{loc}_a \ldots \text{loc}_{a'}\ \langle\text{Click(main, clear)}\rangle \mid \\ & \text{loc}_b \ldots \text{loc}_{b'}\ \langle\text{Click(main, menu)}\rangle \mid \\ & \text{loc}_c \ldots \text{loc}_{c'}\ \langle\text{Back(main)}\rangle \mid \\ & \text{loc}_d \ldots \text{loc}_{d'}\ \langle\text{Swipe(main, left)}\rangle \mid \\ & \varepsilon\end{aligned}$$

(a) *Code Grammar*

$$\begin{aligned}\langle\text{main}\rangle ::= {} & \langle\text{Click(main, clear)}\rangle \mid \\ & \text{loc}_b\ \langle\text{Click(main, menu)}\rangle \mid \\ & \text{loc}_c\ \langle\text{Back(main)}\rangle \mid \\ & \langle\text{Swipe(main, left)}\rangle \mid \\ & \varepsilon\end{aligned}$$

(b) *UI Grammar*

Figure 42: Translating a production rule to target specific lines of code (e.g. loc_b and loc_c)

a depth-first search strategy. We opted for transition nonterminals as they encode sufficient information to produce an event, without requiring changes in the grammar expansion algorithm. The result of this approach are inputs such as: ⟨start⟩ → ⟨Type(auth, PIN, ‘abcd’)⟩ → ⟨Click(auth, clear)⟩ → ⟨Back(auth)⟩ → ⟨Type(auth, PIN, ‘1234’)⟩ → ⟨Click(main, (menu)⟩ → ⟨Back(main)⟩ → ⟨Swipe(main, left)⟩.

By guiding the input generation towards terminal symbols, our inputs avoid repeating the same line of code on different widgets and states.

Nevertheless, one may want only to test specific lines of code. Thus it is possible to rewrite the grammar to remove non-relevant code segments. We guide input production towards specific lines of code using the same approach we used to guide input production towards specific actions on a UI grammar; that is, we rewrite our grammar while removing all non-relevant segments from it. For example, consider the code grammar segment shown in Figure 42a. This production rule reaches multiple lines of code (loc_x). If only loc_b and loc_c are of interest, we can rewrite this production without the remaining code segments (Figure 42b) while retaining all possible transitions, and use it as a regular code grammar to produce test cases.

7.5 Evaluation

The *UI grammar* aims to be a unifying model, encompassing both the graphical elements from the app and its textual inputs. Moreover, it aims to connect these elements with the source code segments they trigger. We previously introduced the concept of UI grammars, proposed an algorithm to mine them from existing test cases automatically, and showed how to use them to produce new tests. In the remainder of this section, after describing our experimental setup, we present our experiments to answer the following research questions:

RQ1 (Grammar mining). Can UI grammars model test cases produced by a random test generator?

RQ2 (UI Grammar Accuracy). Are UI grammars accurate models to guide test generation towards specific app actions?

RQ3 (UI Grammar Expansion). Can mined UI grammars be used to trigger efficiently all actions from the test cases they were mined from?

RQ4 (Code Grammar Accuracy). Are code grammars accurate models to guide test generation towards specific code locations on the app?

RQ5 (Code Grammar Expansion). Can mined code grammars be used to trigger efficiently all lines of code triggered by the test case they were mined from?

7.5.0.1 Modeling Widgets and States

Before mining UI grammars, it is necessary to determine how to identify widgets and UI states uniquely. Natively, Android does not provide any identifier for either. Therefore, several heuristics were proposed to identify states and widgets on Android apps [125, 53, 44]. We adopted the definitions from our previous works [53, 68] for our experiments, as they were already implemented in the test generator that we use in our evaluation. Nevertheless, any other widget and UI state definition could have been used alongside UI grammars.

Widgets: Without source code access, there is no unique identifier for a UI element. The closest information it has is a *resource ID*, which is specified by the developer. Nevertheless, the *resource ID* is optional and seldom used and can also be reused in the same state. Using the Android accessibility service[25], one can obtain a set of properties for each widget w. We used this information to identify a widget (id_w) uniquely by:

$$textid_w = \begin{cases} \text{hint text, content desc., or resource ID} & \text{if } w \text{ is an input field} \\ \text{hint text, text, and content desc.} & \text{if } w \text{ has textual content} \\ \text{resource ID} & \text{if } w \text{ has a resource ID} \\ \text{class name, package name, and fallback ID}^{26} & \text{otherwise} \end{cases}$$

States: As with widgets, Android does not provide a reliable way to identify UI states. While one may consider each activity as a state, Android supports a single-activity multiple-fragment architecture, where the developer loads at runtime different fragments in the same activity. We, thus, opted to define app states from a user perspective based on the widgets available on the screen and their current configuration. They defined a UI state as

$$id_s = \bigcup_{w}^{\mathrm{W}} id_w$$

where id_w is the unique identifier of a widget and W are all the relevant widgets on the screen. Their metric considers as relevant the widgets that have no children or have textual content or are actionable. Moreover, to be resilient against external sources, they ignore elements that do not belong to the app under test. This includes advertisement containers and other apps launched through intents[27] when determining a UI state.

7.5.1 Experimental Setup

Due to the absence of existing test cases for a large corpus of apps to use as a benchmark, we instead evaluate our approach on test cases that are generated by an automated test generator

[25] `https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo`

[26] The fallback ID is based on the position of the widget on the UI hierarchy tree produced by the Accessibility Service, calculated using the parent ID and the index from the current element and all its ancestors until the UI hierarchy root.

[27] Non-app widgets are filtered out based on their package name.

for Android. This is a common approach to create test cases [126], and the generated tests are conceptually similar to a user interacting with the app. That is, the test generator inspects the contents of a screen, chooses a widget to interact with, and interacts with the widget using the accessibility app API. We, thus, expect our results to be valid for realistic existing test cases created by other approaches.

We executed our experiments on physical Google Pixel XL devices and emulators using the same device configuration. Both the emulators and devices use Android 9 (API 28) with all security patches until August 2019. To prevent divergences on the results from the use of physical devices and emulators, each app was tested entirely on a single device or emulator.

We randomly selected 46 Android apps[28] from *F-droid*, an open-source repository of Android apps. We opted for open-source Android apps as they tend to be smaller, and test generators can cover larger amounts of functionality. Note that this is not a limitation from the use of grammars, but instead of the underlying test generation approach used to generate the initial test cases. One can apply UI grammars on larger apps by generating the initial test manually or by using approaches tailored to specific apps. Since our test case is automatically generated, we measure its code coverage to determine its overall quality [61]. We measure code coverage through Jimple statements [63], an intermediate bytecode-like representation of their compiled source code. On average, the apps on the dataset contain 14,389 statements, with the smallest app containing 1,347 and the largest 72,056.

7.5.2 RQ1: Mining Grammars

For this experiment, we executed DM-2 [53] to randomly explore each application for 1000 actions ($\approx$60 minutes), with a forced restart every 100 actions to allow the test generator to explore different paths. We enabled the following action types during the exploration: click, long click, swipe (left, right, up, and down), type, press back, and press home.

We opted for DM-2 as it provides unique identifiers (UID) for each widget and UI state it encounters during testing. It also includes an execution log containing all actions executed during testing. Moreover, we opted for a fixed number of clicks instead of a fixed time for a better comparison with the grammar, which can be used to generate only sequences of actions. Our initial tests had an average Jimple coverage of 39.14%, in line with contemporary black-box testing techniques [29].

Before converting our *initial test cases* into a UI grammar, we preprocessed them to address some of DM-2's specificities. First, we removed non-UI actions, such as *enabling WiFi* or *muting the phone*, as those actions are not relevant for the UI grammar, but are performed internally by DM-2 during each app reset. Moreover, DM-2 considers all non-app states as equivalent—it always presses the back button—but produces different identifiers for each one. To address this limitation, we replace the source and destination states UID of all states which do not belong to the app by a single identifier.

We then used our grammar mining algorithm on the preprocessed DM-2 test traces to extract UI grammars. We were able to successfully extract a UI grammar for all the 46 test traces.

[28]The full list of apps is available in `https://github.com/uigrammar/uigrammar`

Figure 43 shows the size of each of the mined grammars with respect to the number of production rules, and unique states.

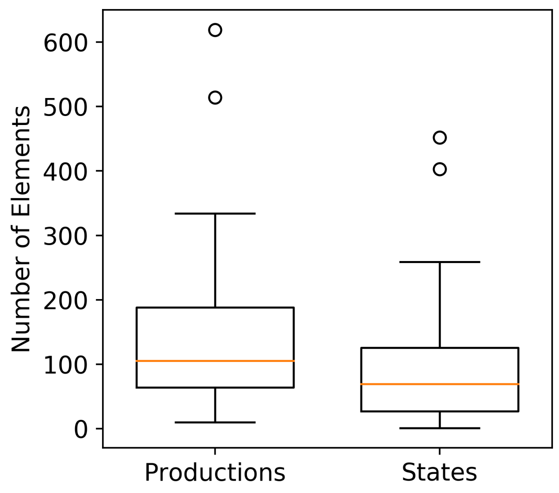


Figure 43: Size of the mined UI grammars measured by number of production rules, unique states and unique widgets

On average, the mined UI grammars were composed of 141 production rules with a minimum of 10 and a maximum of 619. Considering states, the mined UI grammars represent, on average, 93 unique states per test case, with a maximum of 452 states. The largest grammar was produced by the test case from *BeHe Pro*[29] with 619 production rules. An example of the smallest mined UI grammar for the app ALSA Mixer WebUI[30] is shown in Figure 44.

$$
\begin{aligned}
\langle \text{start} \rangle &::= \langle \text{s00} \rangle \\
\langle \text{s00} \rangle &::= \varepsilon \mid \text{Click(w00)}\ \langle \text{Click(s00, w00)} \rangle \\
\langle \text{Click(s00, w00)} \rangle &::= \langle \text{s01} \rangle \\
\langle \text{s01} \rangle &::= \varepsilon \mid \text{Click(w01)}\ \langle \text{Click(s01, w01)} \rangle \mid \\
&\ \text{Click(w02)}\ \langle \text{Click(s01, w02)} \rangle \mid \\
&\ \text{Click(w05)}\ \langle \text{Click(s01, w05)} \rangle \\
\langle \text{Click(s01, w01)} \rangle &::= \langle \text{s00} \rangle \\
\langle \text{Click(s01, w02)} \rangle &::= \langle \text{s02} \rangle \\
\langle \text{Click(s01, w05)} \rangle &::= \langle \text{s03} \rangle \\
&\ \text{(14 more production rules} \ldots \text{)}
\end{aligned}
$$

Figure 44: Segment of the UI Grammar mined for the app ALSA Mixer WebUI

We inspected the reasons behind large grammars (outliers). DM-2's UID heuristic causes a high number of production rules. It uniquely identifies a UI state as the concatenation of all leaf widgets with either a resource ID or text on the screen. When a widget is created, deleted, moved, or its text changes, DM-2 identifies the set of UI elements as a new state. While such

[29] `https://f-droid.org/de/packages/com.vlath.beheexplorer/`
[30] `https://f-droid.org/de/packages/cz.jiriskorpil.amixerwebui/`

an approach allows DM-2 to guide its test towards least interacted UI elements, it also identifies many similar states with different UIDs. Thus, the grammars generated from such tests are closely related to the test itself and cannot be easily generalizable for other tests on the same app. UI grammars, however, are not tied to any specific test generator. They can be mined from other tools, with different state identification mechanisms, or be manually written.

UI grammars could be automatically mined from all test cases generated for 46 open-source apps. The test cases, and resulting grammars contained the following action types: click, long click, swipe (left, right, up, and down), type, press back, and press home.

7.5.3 RQ2: UI Grammar Accuracy

We showed that UI grammars could be mined automatically from existing test cases. However, we have not yet measured how accurate such grammars are. In this research question, we evaluate the grammar accuracy by checking if tests produced from UI grammars can guide test generation towards specific actions.

For this experiment, we reuse the apps, environment, and mined grammars from Section 7.2. First, we randomly selected a statistically significant number of actions as targets. To determine the number of samples necessary, we considered the unique actions in the grammar as population, established a desired confidence level of 90%, a margin of error of 10%, and assumed that the elements follow a normal distribution.

Since the apps in our dataset have different sizes, and the initial tests reached different amounts of coverage, the number of samples required for each app varied between 9 and 69, with an average of 54. For each target, we rewrote the grammar to guide the test towards the specific action, and we generated ten test cases, to explore multiple ways to reach it.

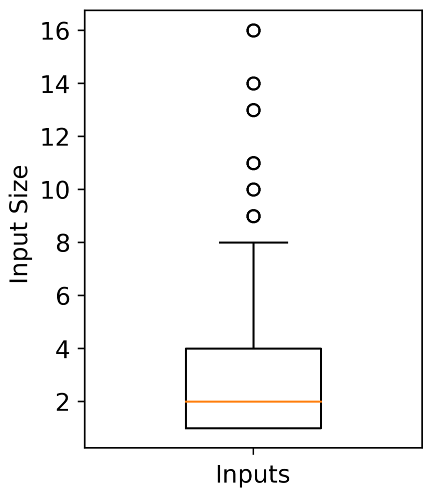
Fuzzing UI grammars that target specific actions resulted in 30,710 sets of inputs, comprising 85,348 actions (including app launch). The produced inputs had, on average, four actions, with a minimum of 1 and a maximum of 16, their distribution is shown in Figure 45a.

We then developed a DM-2 extension named DroidGram to execute our grammar-based test cases. DroidGram consumes a sequence of grammar symbols and decodes them into device actions, while monitoring which grammar symbols are successfully reached. We use this information to measure how often the test inputs can reach their final action.

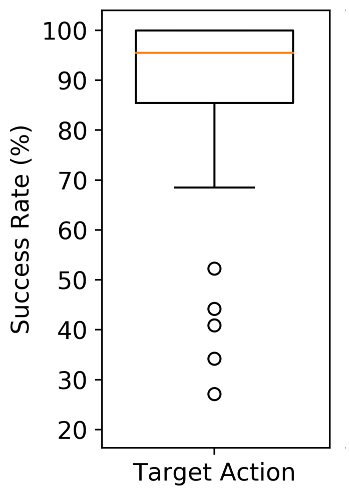
In our experiments, 82% of the inputs generated by fuzzing with UI grammars reached their target action, with a few reaching all their targets and the worst reaching 20.3% of them, as summarized in Figure 45b. These values highlight a limitation of using context-free grammars to model apps: apps are context-sensitive, and some actions can only be triggered under specific contexts. Nevertheless, the same limitation also occurs on approaches that model apps as FSMs.

Most inputs generated by the grammars for *FOSSASIA 2017*[31] and from *NewPipe*[32] failed to reach the targets. We manually inspected these apps to determine the cause. Both apps relied heavily on Internet content and were affected by larger delays during replay; that is, DM-2 could

[31]`https://f-droid.org/wiki/page/org.fossasia.openevent`
[32]`https://f-droid.org/de/packages/org.schabi.newpipe/`



(a) *Input size*



(b) *Reached actions*

Figure 45: Size of inputs used to measure the grammar accuracy and success rate of these inputs reaching their targets

not perform several widget actions because the target widget was not yet on the screen. Our UI grammars does not support modeling communication with external servers.

When producing tests to trigger specific app actions, 82% of the test cases produced by a mined UI grammar successfully reach their target.

7.5.4 RQ3: UI Grammar Expansion

In this section, we evaluate if mined UI grammars can be used to effectively re-trigger all UI actions from the test case it was mined from.

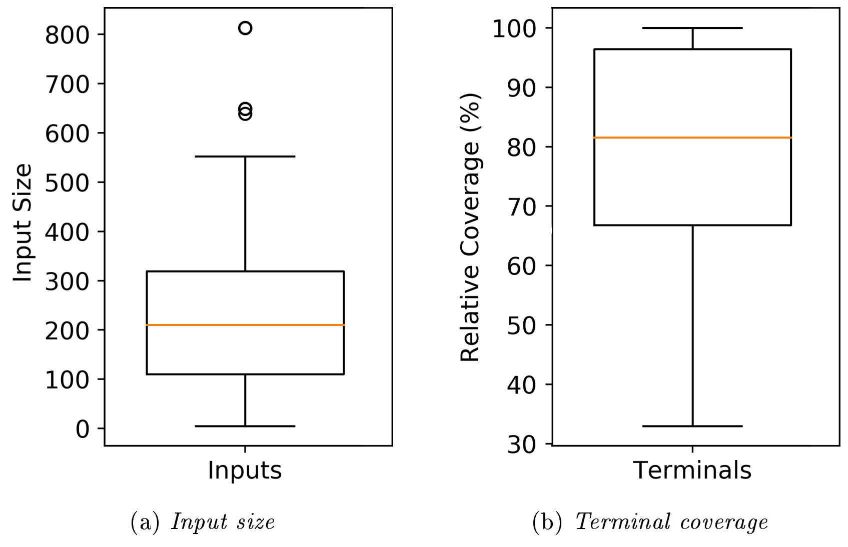
For this experiment, we reuse the apps, environment, and mined grammars from Section 7.2. We fuzz with each grammar to produce inputs that cover all UI actions (terminal symbols) in the grammar and concatenate them into a single test case. While multiple inputs may be necessary to cover all terminals in the grammar, each input starts with a *restart app* action (⟨start⟩). Thus, we have a test case in which the app restarts multiple times by concatenating the inputs.

We generate ten different test cases for each app to explore multiple paths within the grammar, resulting in 460 inputs. We used DroidGram to execute our grammar-based test cases while monitoring which grammar terminals and lines of code are successfully reached. We measured grammar and code coverage relative to the original test suite; that is, how many different actions (terminals) and the inputs produced by the grammar can reach lines of code from the original test case.

The test cases produced by our UI grammars required, on average, 229 actions to re-trigger all the actions from the original test case (≈23%), as shown in Figure 46a. The longest grammar-based test cases were 812 actions long (83% of the original) and were produced by the grammar from RedReader[33], a Reddit client. We manually inspected the app and grammar and observed

[33]`https://f-droid.org/de/packages/org.quantumbadger.redreader/`

that this app has several UI elements with textual content, which are created dynamically. DM-2 identified them as different UI states resulting in a grammar where most state nonterminals contain a single possible action, allowing the test generation undermining our terminal guided input generation approach.



(a) *Input size* (b) *Terminal coverage*

Figure 46: Size of the test cases produced by UI grammars, and their terminal coverage relative to the initial grammar

We summarize our results concerning terminal (action) coverage in Figure 46b. Test cases produced by the UI grammar triggered, on average, 70% of the actions, with a minimum of 16%. Additionally, 62 inputs—13.5% of all inputs produced—reached 100%. While most grammar-based test cases reached most actions (terminals) from the original test case, some inputs were able to trigger meager amounts. We manually inspected the worst performing app.

The *BART Runner*[34] app, which checks public transportation schedules in San Francisco, obtained the worst results. Its functionality is highly dependent on external resources, such as train schedules and the current date and time, which are not modeled by the UI grammar. Since DM-2 identifies a widget based on their textual content, the inputs produced by the grammar were not valid, as different connections are shown at each time. Such a problem, however, occurs with any test case that identifies widgets based on their textual content and can be mitigated by using other metrics to define widgets uniquely.

Regarding code coverage, grammar-based test cases reached an average of 83% of the lines of code reached by the original test case, with a minimum of 14.7% and 30 (6.5%) of them reaching 100%, as shown in Figure 47. The *BART Runner* app, again, achieved the worst code coverage results, as it could not execute large parts of the input.

On an action-per-action comparison, UI Grammar-based test cases are more effective at achieving code coverage than the original test case. On average, they perform better than the initial test on the first ≈170 actions, with the largest difference being observed at ≈70 actions. Until ≈210 actions, both tests are virtually tied, and after 300 actions, the original test case significantly outperforms UI grammar-based test cases. Nevertheless, the same applies to any

[34] `https://f-droid.org/de/packages/com.dougkeen.bart/`

systematic approach as, given enough time, a random test generator outperforms any systematic approach [127]. Note as well that most test cases are shorter than 300 actions and, thus, would no longer be executing at that moment.

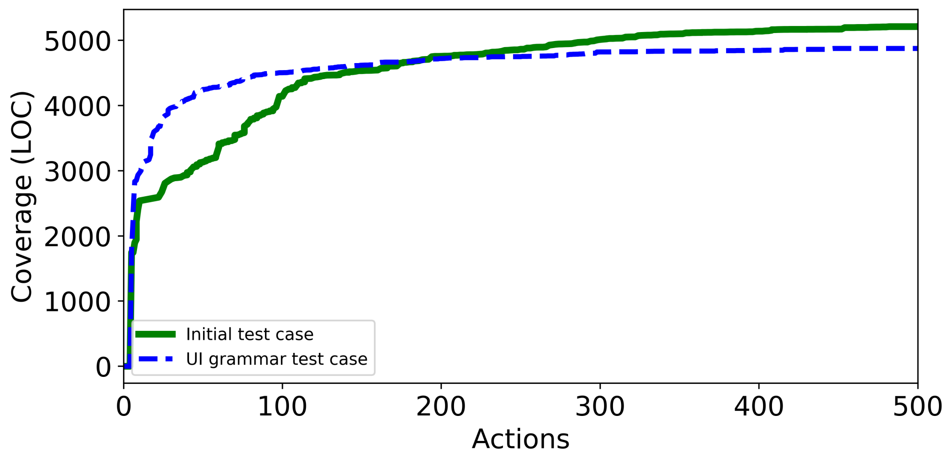


Figure 47: Average code coverage over time for UI grammar-based test cases, compared to the the original test case on the first 50% of the original actions

With 23% of the original actions and only a few outliers, most grammar-based test cases are significantly shorter than the test case they were mined from. Thus, we envision such tests as complementary to the original test case. One can execute them to detect errors faster on large swaths of the actions (80%) and execute the complete and more time-consuming test-suite after such tests succeed.

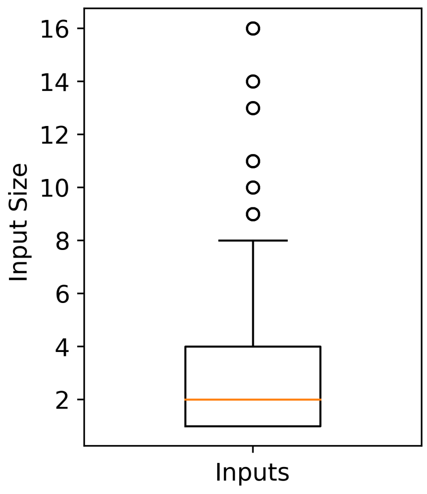
UI grammar-based test cases reach 80% of the original test case actions with 23% of the input size. Moreover, they are more effective than the original test case at achieving code coverage on short tests.

7.5.5 RQ4: Code Grammar Accuracy

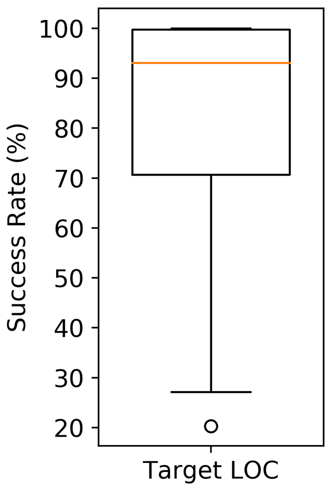
We showed how mined UI grammars successfully guide test generation towards the actions from the test case they were mined from. In this section, we evaluate the accuracy of code grammars by checking if tests produced from code grammars can guide test generation towards specific lines of code.

For this experiment, we reuse the apps, environment, and initial test cases from Section 7.5.2. During our initial experiments, we collected the lines of code that were reached by each action in the original test suite. We attach this information to DM-2's trace and use it to mine code grammars, as we showed in Section 7.4. The code grammars are identical in the number of productions and nonterminal symbols to the UI grammars generated in Section 7.5.2, varying only in the number of terminal symbols.

Similar to our experiment in RQ2 (Section 7.5.3), we randomly select a statistically significant number of terminals from our code grammars as targets. While in RQ2, we considered as a population the unique actions in the grammar This time, we consider as a population the unique



(a) *Input size*



(b) *Reached actions and code segments*

Figure 48: Input size and coverage achieved with code grammar-based test cases

lines of code reached by the original test. We again established the desired confidence level of 90%, a margin of error of 10%, and assumed that the elements follow a normal distribution.

Since the apps in our dataset have different sizes, and the initial tests reached different amounts of coverage, the number of samples required for each app varied between 55 and 69, with an average of 67. For each target (line of code), we rewrote the grammar to guide the test towards it, and we generate ten test cases, to explore multiple ways to reach it.

Fuzzing code grammars that target specific lines of code resulted in 30,710 sets of inputs, comprising 85,348 actions (including app launch). The produced inputs had, on average, four actions, with a minimum of 1 and a maximum of 16. Their distribution is shown in Figure 48a.

We reused DroidGram to execute the grammar-based test cases and monitor both the reached actions and Jimple coverage of this new test. We use this information to evaluate how often the test inputs can *trigger their target line of code.*

70.5% of the inputs produced by a code grammar can reach their target line of code, with the worst-performing seed reaching only 4%, as summarized in Figure 48b. These values are slightly below the 82% success in reaching an action using a UI grammar, further highlighting the limitations of using a context-free model to represent apps: certain lines of code can only be reached in specific app contexts. Nevertheless, this is not only a limitation of UI grammars but also occurs approaches that model apps as FSMs.

Most inputs generated from the grammars for *FOSSASIA 2017*[35] and *NewPipe*[36] could not successfully reach the target actions nor lines of code. We manually inspected these apps to determine the cause.

Both apps relied heavily on Internet content and were affected by larger delays during replay; that is, DroidGram could not perform several widget actions because the target widget was not yet on the screen. Neither our UI nor code grammars, however, support modeling the

[35]`https://f-droid.org/wiki/page/org.fossasia.openevent`
[36]`https://f-droid.org/de/packages/org.schabi.newpipe/`

communication with external servers.

When producing tests to trigger specific code locations, 70.5% of the test cases produced by a mined code grammar successfully reach their target line of code.

7.5.6 RQ5: Code Grammar Expansion

We previously showed how mined code grammars successfully guide test generation towards specific lines of code. In this section, we evaluate if mined code grammars can be used to effectively re-trigger all lines of code from the test case it was mined from.

For this experiment, we reuse the apps, environment, and code grammars from Section 7.5.5. We then fuzz with each code grammar to produce ten different test cases for each app to explore multiple paths within the grammar and execute the resulting 460 test cases with DroidGram while monitoring the covered actions and lines of code.

Test cases produced by our code grammars required, on average, 107 actions to reach all lines of code from the original test suite, with a maximum of 812 actions. This represents ≈11% of the original test suite size and ≈50% of the actions required to trigger all terminals compared to the UI grammar (7.5.4). The input size distribution is shown on Figure 49.

As in Section 7.5.4, we observed some outliers regarding the input size with the same RedReader app having the longest input. The causes are the same as those in we presented in Section 7.5.4: since most production rules on its grammar contain a single possible expansion, the input generation strategy was not able to guide the testing towards more efficient inputs properly.

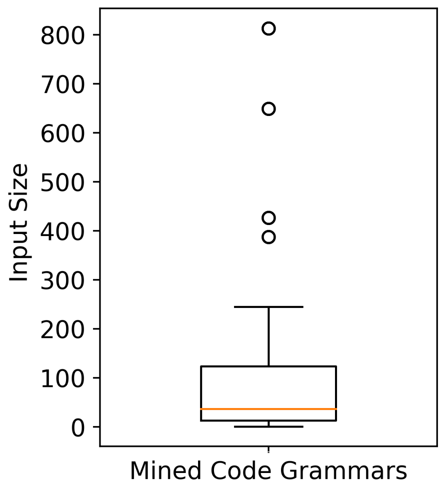


Figure 49: Size of the test cases produced by code grammars

We summarize our results concerning code coverage in Figure 50. Test cases produced using the code grammars reached, on average, 79% of the lines of code from the original test suite–most of which is obtained with less than 60 actions–with the worst input reaching 14.3% and two inputs reaching 100%. The worst inputs were produced by the *Simple Last.fm* app[37]. In this app, most lines of code covered by the initial test were concentrated in a few grammar productions,

[37]`https://f-droid.org/de/packages/com.adam.aslfms/`

which could not be reached by all test cases due to app context, resulting in significant coverage loss.

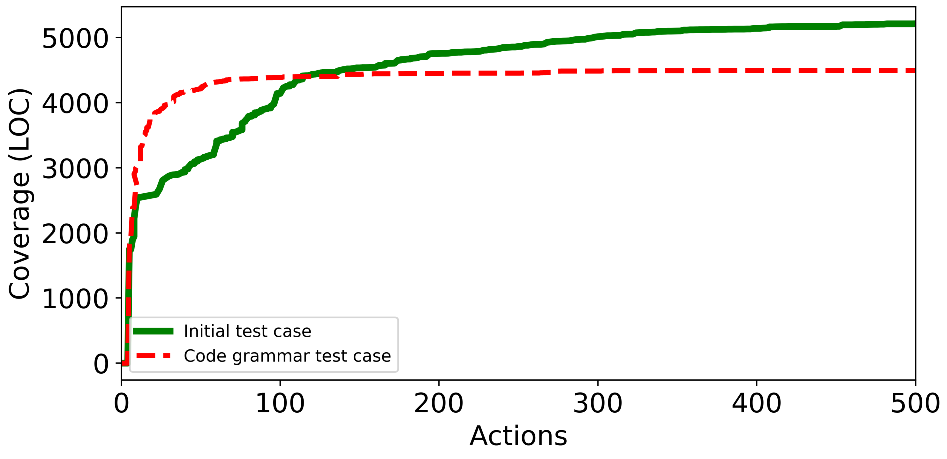


Figure 50: Code coverage over time for UI grammar-based test cases, compared to the the original test suite on the first 50% of the original actions

On a per action basis, tests using code grammars are faster at achieving code coverage than the original test suite, as shown in Figure 50. On average, they perform better than the original test on executions of up to 100 actions, with the largest difference being observed at less than 50 actions. Until ≈130 actions, both tests are virtually tied, and after 150 actions, the original test case significantly outperforms code grammar-based test cases. Nevertheless, the same applies to any systematic approach as, given enough time, a random test generator outperforms any systematic approach [127]. Note as well that most test cases are significantly shorter than 150 actions and, thus, would no longer be executing at that moment.

When compared to test cases produced by UI grammars, tests produced using a code grammar reach ≈95% of the coverage, with 50% of the actions. Therefore, the same benefits apply to test cases based on code grammar. Such tests can complement the original test suite, allowing errors to be detected quickly on 79% of the code. At the same time, complete and more time-consuming test-suite can be executed after such tests succeed.

Test cases produced using mined code grammars obtain code coverage with less actions than the tests the grammars were mined from, reaching 79% of the code with less than 10% of the actions.

7.5.7 Threats to Validity

In this section, we discuss the limitations and threats to the validity of our technique and evaluation.

Regarding external validity, we cannot ensure that our results generalize to all apps and testing tools, due to the size of our dataset and automatically generated test cases. We randomly selected our apps from an open-source (F-Droid) repository, which has been previously used to evaluate test generators [29, 68, 82]. While we evaluated our approach on Android apps, the concepts presented in this paper also apply to other platforms or domains (e.g., iOS and web applications). Therefore, more evaluations of other tools and platforms would be needed to mitigate these threats.

Regarding internal validity, we used DM-2 to generate the test case from which we mined the UI grammars and, thus, we inherit its limitations. If DM-2 misidentified a state or failed to identify a widget, the same will be reflected in the mined grammars. Additionally, DM-2 cannot ensure that all test cases it generates can be re-executed, due to the inherent flakiness of Android UI tests: an app may reach different states depending on external conditions, such as time and location, altering its typical execution path. They may also communicate with external servers, which may take longer than usual to reply. Furthermore, the mined grammars can only cover behaviors seen on the initial test. The mined grammars will not know any behavior not contained in the test cases. UI grammar can, however, be mined from any sequences of events with uniquely identifiable widgets and states, be it automatically generated or manually written.

Regarding construction validity, UI grammars only model interactions that happen through the UI (user events). While the grammar encoding can be used to model system events, such as call/message received, such an implementation has not been evaluated. Moreover, we used a semi-random testing approach to generate our initial test case. We opted for it instead of manually written tests because they are not always available and may cover only specific app functionality. With randomly generated tests, we avoid specifying which app features to test. We also did not use more advanced test generation techniques as they cannot be applied without instrumenting the OS or app [46, 128]. Our evaluation showed that UI grammars could reach the majority of the original actions and code segments with a much smaller number of actions. Such numbers may vary when comparing to manually written tests or different test generation techniques.

Finally, we use context-free grammars as a unifying model. However, many UI actions can only be triggered in specific contexts. In our experiments, we mitigated this problem by using DM-2's UID to identify states and widgets. DM-2's UID mechanism differentiates between different UI contexts through the elements which are available on the screen. It allowed us to reach 82% of the target actions successfully. Nevertheless, DM-2's UID metric still does not model all possible context-sensitive behavior. For example, it produces the same identifier if the widget in the state is enabled—and thus can be interacted with—or not. Moreover, our experiments mining grammars in Section 7.5.2 and expanding them in Section 7.5.4, showed that when the state identification metric is too fine-grained towards context-sensitive behavior, the resulting context-free grammars, as well as the inputs produced by it, become larger.

7.6 Limitations

Our experiments showed that context-free grammars could model both UI transitions and textual elements of an app. Nevertheless, our implementation and approach do not work for all apps. Our implementation inherits the limitations of DM-2, which we previously discussed in Section 3.7.

Another limitation of our implementation is the association between actions and lines of code. We defined the lines of code triggered by an action as those reached when the action occurs. However, an app may start delayed or scheduled tasks, such as starting a download in the background or checking for updates. In our implementation, this functionality will be associated with the action that occurs when it executes, not with the action that started it. While our approach is abstract about the mechanism used to associate lines of code and actions, how to address this problem is still an open question.

Concerning our approach, it relies on context-free grammars and, thus, do not model context. Using a UI grammar to create tests that trigger context-dependent functionality—actions or code—will produce invalid test cases. Moreover, UI grammars model a single app and cannot be used to test, for example, client-server architectures fully.

7.7 Related Work

In Section 2.2, we presented existing test generation approaches, which are loosely related to our work. This section summarizes the most relevant works related to this research.

Model-based input generators encode an app as a model which describes its functionality and use it to generate inputs systematically. Such models can be defined as a priori or automatically inferred. Model-based Android test generators have employed various encodings to model the behavior of an app, such as finite-state models [35], graphs [40], test specifications [129], and custom representations [38].

Regarding our model inference, our approach is similar to *DroidBot* [37], *Android Ripper* [35, 36] and APE [130], which dynamically generate a transition model of an app during testing, based only on the run-time state of its GUI widgets. Other closely related approaches are *Stoat* [44] and A^3E *Targeted* [40], which uses both static and dynamic analysis to infer a stochastic model of the app's GUI and use it for testing. Our approach uses context-free grammars, a well-known formalism, to specify input languages to model apps. Compared to state machines, grammars are more readable and easier to extend and are particularly useful for modeling textual inputs. While our UI grammar can be manually specified or automatically inferred from test executions, their main advantage is using a single representation to model the app UI and its transitions as well as the lines of code reached by each UI element. By combining both representations in a standard formalism we can reuse established input generation techniques [131, 132] to guide test generation towards specific UI actions [133] or code segments [134].

Regarding our model representation, while not frequently used to test Android apps, grammars have been intensively used to test compilers [135, 136]. Techniques such as *Travor* [137] and *Nautilus* [132] employ specific grammar encodings for general-purpose fuzzing. Such techniques could be used without modification to produce inputs from our UI grammar.

Our work in this chapter is conceptually similar to dynamic feature location [138]. In software evolution and maintenance, dynamic feature location attempts to identify software features by comparing execution traces with and without the features [139]; ranking them accordingly to the frequency and location in which specific methods are used [140]; or exploiting execution traces and the source code [141, 142, 143]. UI grammar productions can be associated with code segments and thus used for features location. However, their main advantage, is that they can quickly produce multiple inputs to trigger the same feature without following the sequence of actions from the original test case. Also, each generated grammar input is significantly shorter than the original test suite. Finally, UI grammars can be easily extended and combined with other grammars, allowing new inputs to cover the same features to be added.

7.8 Lessons Learned

To thoroughly test programs via graphical user interfaces, one has to consider both graphical and textual input. User interface grammars encode states, transitions, user interactions, and textual inputs in a single, unified representation. In this chapter, we showed how to use UI grammars to model complex user interactions on the Android platform, how to mine UI grammar from test cases automatically, and how to use UI grammars to produce test inputs targeting specific elements, interactions, and covered code.

In our experiments, we automatically extracted UI grammars out of 46 test cases for different open-source apps. We then used UI grammars to produce test cases targeted towards specific elements in the grammar—that is, towards specific interactions in the app—and 82% of the test cases produced through a UI grammar reached their targets successfully. Finally, using the mined UI grammars to produce test inputs, we covered most of the UI actions (80%), and source code (83%) form the original test case, with significantly shorter test cases.

The use of context-free grammars to model graphical user interfaces brings a solid foundation to the field, opening up several future research possibilities.

- UI grammars allow for much better *modularity and interplay* in test generation tools. Rather than having monolithic tools in which modeling, mining, and testing cannot be separated from each other, we can have future approaches model, mine, or test using one single representation that is well-known and well-understood by programmers and researchers alike.
- User interface grammars allow to directly apply established techniques for grammars from formal languages. For instance, grammars can be easily *converted* from and into alternate language models such as regular expressions or finite-state models.
- Being grammars, UI grammars can also act as *parsers*—for instance, to parse the traces of existing tests and thus recombine fragments of input sequences, say from past bug-inducing inputs [131].
- In a grammar, one can add *probabilities* to individual productions, allowing to focus tests on specific (textual or graphical) features—or on associated code locations. The combination of probabilities and parsing allows for *anomaly detection* over events and transitions from the grammar.

Reproducibility: To facilitate replication and extension, all our work is available as open source. The replication package is available at:

`https://github.com/uigrammar/uigrammar`

Chapter 8

Conclusion and Future Work

In this thesis, we investigated how to learn and model the language of Android apps. Users can effectively use apps because they know this language. They know how to interact with the UI elements and which textual values to input. Our statement in this thesis was that a test generator could also learn this language and, in doing so, would be able to produce better tests. In the first part of this thesis, we presented DM-2, a platform for Android test generation, which worked as the base for the remainder of our approaches.

In the second part of the thesis, we presented our approaches to *learning the language of apps*. We first learned *which UI elements to interact with* by emulating human behavior. When faced with a new app, a human uses its experience (knowledge) from using other apps to know how to interact with it. Similarly, we learned offline, from a set of popular apps from the Google Play Store, a UI interaction model that predicts how likely a UI element is to be interactive. We then extended two test generators to consume our model (Droidmate-M and Droidbot-M) and showed that using such a model lead to significant improvement in code coverage on both test generators.

We then learned the next part of the language of apps, namely: *how to interact with a UI element?* A human uses its previous experience when interacting with an app for the first time. However, once it has used the app, it learns how that specific app behaved, which actions worked and which did not. Thus, we followed a similar approach and learned how to interact with the UI elements of an app while using it. We modeled test generation as an instance of the *MAB problem* and employed the traditional reinforcement learning algorithms used to address it. Our experiments showed that learning how to interact with app UI elements during testing lead to further coverage improvement when compared against reusing only previously gathered knowledge. They also showed that dynamically learned knowledge can be used alongside other testing algorithms, producing tests with higher code coverage.

We then closed the second part of this thesis by learning *which input values should be typed.* Many apps require complex inputs, such as an address, a departure airport, or a product to search for. Humans understand the semantics of the UI and enter such information easily. To adequately interact with such apps, a test generator must also be able to enter such values. With this goal, we presented Saigen to automatically identify the UI semantics and query a knowledge base for candidate input values. Our experiments showed that using such queried inputs lead to an overall improvement in code coverage.

Finally, in the last part of this thesis, we modeled the language of apps, bringing together UI actions, transitions, and textual inputs and source code into a unified model. While the UI actions and transitions can be represented as a regular language, input values do not. Therefore, we used context-free grammars as a unifying formalism. We showed how grammars model test cases, how they could be extracted from existing tests, and how they can be used to produce inputs. Additionally, we showed how to associate grammar productions with the app code they trigger, and exploited this relationship to produce test cases directed towards specific code statements.

We acknowledge that even by learning the language of apps, one cannot ensure an app is completely tested. Some apps—such as games—require precise knowledge to be used. Some app functionality is only available after following a long and complicated sequence of events—such as registering, receiving a confirmation email, accessing a website to enable the account, and returning to the app to log in. Learning such interaction patterns is still an open question.

Throughout this thesis (Sections 3.9, 4.6, 5.7, 6.6, and 7.8), we presented the lessons we learned and highlighted open qu estions for each of the approaches we presented. While this thesis introduced how to learn and model different aspects of app interaction, it is just the beginning.

Learning interaction sequences Some app functionality requires sequences of events to be triggered. To buy an item on a shopping app, for example, it usually is necessary to add the item to the cart, click on checkout, enter billing and shipping information, and confirm the purchase. Learning interaction patterns only makes the test generator more likely to trigger app functionality but does not guarantee that the right functionality is triggered, in the correct sequence. Triggering the right interactions, in the right order, can lead to significant improvements. In Section 6.3, we observed that splitting the widgets into the input and the non-input fields, and interacting with the input fields before the rest, already improves test coverage. Similar behavior is seen in [50] when splitting the UI into buttons and non-buttons. Learning interaction sequences could allow test generators to reach more functionality.

Associating UI semantics When using an app, humans rely not only on their knowledge of how apps work but also on the semantics of its UI. Our learning techniques could benefit from semantic information. UI semantics can be used to determine interaction patterns and be a piece of valuable information for learning interaction sequences. Better mechanisms for semantic extraction can also be used to obtain more accurate real-world values for testing, such as by restricting a *from* label in a flight app to airports. Semantics can also be associated with our mined UI grammars, opening possibilities in fields such as anomaly detection and test transfer.

Learning app functionality We learned how to interact with an app, but not how to use it. That is, we did not explore how to trigger app functionality correctly. Learning interaction sequences and associating them with UI semantics could be used to allow test generators to reuse previously acquired knowledge when testing different apps. Such behavior could effectively allow the development of approaches that learn how to use apps, not only from an interaction perspective but by actually learning how to trigger app functionality in the process.

Finally, the general concepts we have presented in this thesis are not limited to the Android platform or mobile apps in general. Our techniques to learn and model the language of apps can be used on different platforms, such as web or iOS. Also, while we designed DM-2 to test Android apps, its architecture is easily extensible. One can, for example, extend its automation engine to interact with iOS apps and reuse out-of-the-box the approaches we implemented. We conclude this thesis in the hope that it provides useful insights on how to teach test generators to use apps effectively, leading to better tests.

Curriculum Vitae

Nataniel Pereira Borges Junior was born in Itajaí, Brazil on November 1st, 1989. He received his bachelor's degree in information systems from Federal University of Santa Catarina, Brazil in 2014. In 2017 he received his master's degree in computer science from Saarland University, Germany and became a PhD student at the CISPA–Helmholtz Center for Information Security. Since May 2019, Nataniel is an Engineer/Researcher at Google.

Publications

1. Nataniel P Borges Jr. Data flow oriented ui testing: exploiting data flows and ui elements to test android applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 432–435, 2017

   Contributions: (I) original idea;

2. Nataniel P Borges Jr, Jenny Hotzkow, and Andreas Zeller. Droidmate-2: a platform for Android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 916–919. ACM, 2018

   Contributions: (I) original idea of an extensible testing platform; (II) idea and development of the exploration strategies; (III) idea and development of the monitoring proxy; (IV) original idea of extensible model features; (V) partial evaluation.

3. Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 133–143. ACM, 2018

   Contributions: (I) original idea of an mining an interaction model from a crowd of apps and using it for testing; (II) idea and development of the exploration strategy; (III) idea and development of the fitness boost; (IV) partial evaluation.

4. Jie Huang, Nataniel P Borges Jr, Sven Bugiel, and Michael Backes. Up-to-crash: Evaluating third-party library updatability on android. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30. IEEE, 2019

   Contributions: (I) in depth evaluation with DM-2.

5. Nataniel P Borges Jr and Andreas Zeller. Why does this app need this data? automatic tightening of resource access. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 449–456. IEEE, 2019

   Contributions: (I) original idea; (II) implementation; (III) evaluation.

6. Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 296–306. ACM, 2019

   Contributions: (I) original idea of using reinforcement learning to learn app interactions; (II) original idea of using a statically mined UI model as previous knowledge; (III) idea and development of the fitness proportionate selection hybrid; (IV) partial evaluation.

7. Mohammad Naseri, Nataniel P Borges Jr, Andreas Zeller, and Romain Rouvoy. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. *Proceedings on Privacy Enhancing Technologies*, 2019(2):291–305, 2019

   Contributions: (I) original idea for detection with static analysis; (II) partial evaluation.

8. Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P Borges Jr, Eric Bodden, and Andreas Zeller. Heaps'n Leaks: How heap snapshots improve android taint analysis. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020

   Contributions: (I) partial evaluation.

9. Tanapuch Wanwarang, Nataniel P Borges Jr, Leon Bettscheider, and Andreas Zeller. Testing apps with real-world inputs. In *Proceedings of the 1st IEEE/ACM International Conference on Automation of Software Test (AST)*. ACM, 2020

   Contributions: (I) original idea; (II) theoretical foundation; (III) partial implementation; (IV) partial evaluation.

10. Nataniel P Borges Jr, Manuel Benz, Eric Bodden, and Andreas Zeller. Testing with user interface grammars. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM, 2020. **(Manuscript under review with number TOSEM-2020-0062)**

    Contributions: (I) original idea; (II) partial implementation; (III) evaluation.

11. Nataniel P Borges Jr, Jenny Rau, and Andreas Zeller. Speeding up gui testing by on-device test generation. In *Proceedings of the 35rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2020

    Contributions: (I) original idea; (II) partial implementation; (III) evaluation.

Bibliography

[1] StatCounter. Gartner says global smartphone sales continued to decline in second quarter of 2019. https://www.gartner.com/en/newsroom/press-releases/2019-08-27-gartner-says-global-smartphone-sales-continued-to-dec, August 2019. Accessed: 2019-09-11.

[2] Gartner. Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, August 2019. Accessed: 2019-09-11.

[3] Statista. Number of available applications in the google play store from december 2009 to june 2019. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, August 2019. Accessed: 2019-09-11.

[4] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[5] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play?: a large-scale empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 231–242. ACM, 2018.

[6] Luis Cruz, Rui Abreu, and David Lo. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering*, pages 1–31, 2019.

[7] Bram Adams and Shane McIntosh. Modern release engineering in a nutshell–why researchers should care. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 5, pages 78–90. IEEE, 2016.

[8] Mishaal Rahman. The sorry state of android fragmentation: An example to understand developers' plight. https://www.xda-developers.com/the-sorry-state-of-android-fragmentation/, 2016s. Accessed: 2019-10-02.

[9] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability*, (99):1–22, 2018.

[10] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[11] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[12] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[13] Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. Combining automated gui exploration of android apps with capture and replay through machine learning. *Information and Software Technology*, 105:95–116, 2019.

[14] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Link: exploiting the web of data to generate test inputs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 373–384. ACM, 2014.

[15] Google Inc. Testing support library. `https://developer.android.com/topic/libraries/testing-support-library/index.html`, 2016. Accessed: 2016-10-19.

[16] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237, 2016.

[17] Cyanogen Inc. Cyanogen os. `https://www.cyanogenmods.org/`, August 2019. Accessed: 2019-09-11.

[18] Lineage. Lineageos android distribution. `https://lineageos.org/`, August 2019. Accessed: 2019-09-11.

[19] Google. Espresso. `https://developer.android.com/training/testing/espresso/`, 2019. Accessed: 2017-10-12.

[20] Android. monkeyrunner. `https://developer.android.com/studio/test/monkeyrunner`, 2019. Accessed: 2019-10-17.

[21] RobotiumTech. User scenario testing for android. `https://github.com/RobotiumTech/robotium`, 2019. Accessed: 2019-10-17.

[22] JS Foundation. Appium: Mobile app automation made awesome. http://appium.io/, 2019. Accessed: 2019-10-17.

[23] Roboletric. Roboletric: Test drive your android code. http://robolectric.org/, 2019. Accessed: 2019-10-17.

[24] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 72–81. IEEE, 2013.

[25] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 349–366, 2015.

[26] Yongjian Hu and Iulian Neamtiu. Fuzzy and cross-app replay for smartphone apps. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 50–56, 2016.

[27] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM, 2014.

[28] Kimberly Tam, A L I Feizollah, N O R Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The Evolution of Android Malware and Android Analysis Techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41, 2017.

[29] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for Android: Are we there yet? (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE, 2015.

[30] Google Inc. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html, 2017. Accessed: 2017-06-19.

[31] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[32] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards black box testing of android apps. In *2015 10th International Conference on Availability, Reliability and Security*, pages 501–510. IEEE, 2015.

[33] Chao Yang, Guangliang Yang, Ashish Gehani, Vinod Yegneswaran, Dawood Tariq, and Guofei Gu. Using provenance patterns to vet sensitive behaviors in android apps. In *International Conference on Security and Privacy in Communication Systems*, pages 58–77. Springer, 2015.

[34] Konrad Jamrozik and Andreas Zeller. DroidMate: a robust and extensible test generator for android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 293–294. ACM, 2016.

[35] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261. ACM, 2012.

[36] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[37] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for Android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 23–26. IEEE, 2017.

[38] Wontae Choi, George Necula, and Koushik Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, volume 48, pages 623–640. ACM, 2013.

[39] Inês Coimbra Morgado and Ana CR Paiva. Testing approach for mobile applications through reverse engineering of ui patterns. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 42–49. IEEE, 2015.

[40] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Acm Sigplan Notices*, volume 48, pages 641–660. ACM, 2013.

[41] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in gui testing of android applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 559–570. IEEE, 2016.

[42] Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.

[43] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, and Xiaorui Gong. SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 93–104, 2012.

[44] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256. ACM, 2017.

[45] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*, pages 899–914. IEEE, 2015.

[46] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[47] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[48] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE'14*, pages 599–609, 2014.

[49] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[50] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. CuriousDroid: automated user interface interaction for Android application analysis sandboxes. In *International Conference on Financial Cryptography and Data Security*, pages 231–249. Springer, 2016.

[51] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–268, 2018.

[52] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 33–44. IEEE, 2016.

[53] Nataniel P Borges Jr, Jenny Hotzkow, and Andreas Zeller. Droidmate-2: a platform for Android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 916–919. ACM, 2018.

[54] Nataniel P Borges Jr and Andreas Zeller. Why does this app need this data? automatic tightening of resource access. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 449–456. IEEE, 2019.

[55] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[56] Ting Su. Fsmdroid: guided gui testing of android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 689–691. IEEE, 2016.

[57] Marvin Wißfeld. *ArtHook: Callee-side Method Hook Injection on the New Android Runtime ART*. PhD thesis, Saarland University, 2015.

[58] Larissa Azevedo, Altino Dantas, and Celso G Camilo-Junior. Droidbugs: An android benchmark for automatic program repair. *arXiv preprint arXiv:1809.07353*, 2018.

[59] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. Droidleaks: a comprehensive database of resource leaks in android apps. *Empirical Software Engineering*, pages 1–49, 2019.

[60] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 560–564. IEEE, 2015.

[61] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82. ACM, 2014.

[62] R Winsniewski. Apktool: a tool for reverse engineering android apk files. `https://ibotpeaches.github.io/Apktool`, 2012. Accessed: 2016-07-27.

[63] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[64] Lingfeng Bao, Tien-Duy B Le, and David Lo. Mining sandboxes: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455. IEEE, 2018.

[65] Thomas D White, Gordon Fraser, and Guy J Brown. Improving random gui testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 307–317, 2019.

[66] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines. In *42nd International Conference on Software Engineering (ICSE'20). ACM, New York, NY*, 2020.

[67] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. Poster: Efficient gui test generation by learning from tests of other apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 370–371. IEEE, 2018.

[68] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 133–143. ACM, 2018.

[69] comScore Inc. The 2017 U.S. Mobile App Report. `https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/The-2017-US-Mobile-App-Report`, 2018. Accessed: 2018-02-26.

[70] Android. Android design guidelines. `https://developer.android.com/design/index.html`, 2018. Accessed: 2019-10-17.

[71] Android. Material design for android. `https://developer.android.com/design/material/index.html`, 2018. Accessed: 2019-10-17.

[72] Banga and Weingold. *Essential Mobil Interaction design*, volume 3. 2004.

[73] Khalid Alharbi and Tom Yeh. Collect, decompile, extract, stats, and diff: Mining design pattern changes in android apps. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 515–524. ACM, 2015.

[74] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. Detecting behavior anomalies in graphical user interfaces. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 201–203. IEEE Press, 2017.

[75] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

[76] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[77] Sotiris B Kotsiantis, Ioannis D Zaharakis, and Panayiotis E Pintelas. Machine learning: a review of classification and combining techniques. *Artificial Intelligence Review*, 26(3):159–190, 2006.

[78] PR Freeman. Algorithm as 145: exact distribution of the largest multinomial frequency. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(3):333–336, 1979.

[79] Julia Rubin, Michael I Gordon, Nguyen Nguyen, and Martin Rinard. Covert communication in mobile applications (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 647–657. IEEE, 2015.

[80] Alan Dix. Human-computer interaction. In *Encyclopedia of database systems*, pages 1327–1331. Springer, 2009.

[81] Stuart McIlroy, Nasir Ali, and Ahmed E. Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21(3):1346–1370, Jun 2016.

[82] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 296–306. ACM, 2019.

[83] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. QBE: QLearning-based exploration of Android applications. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*, pages 105–115. IEEE, 2018.

[84] Michael N Katehakis and Arthur F Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987.

[85] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[86] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684, 1957.

[87] Harald Benzing, Karl Hinderer, and Michael Kolonko. On the k-armed Bernoulli bandit: Monotonicity of the total reward under an arbitrary prior distribution. *Mathematische Operationsforschung und Statistik. Series Optimization*, 15(4):583–595, 1984.

[88] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* PhD thesis, King's College, Cambridge, 1989.

[89] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[90] Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning*, pages 437–448. Springer, 2005.

[91] Olivier Chapelle and Lihong Li. An empirical evaluation of Thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257, 2011.

[92] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. A tutorial on Thompson sampling. *Foundations and Trends® in Machine Learning*, 11(1):1–96, 2018.

[93] Sebastian Bauersfeld and Tanja Vos. A reinforcement learning approach to automated GUI robustness testing. In *Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE 2012)*, pages 7–12, 2012.

[94] Sebastian Bauersfeld and Tanja EJ Vos. User interface level testing with TESTAR; what about more sophisticated action specification and selection? In *SATToSE*, pages 60–78, 2014.

[95] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[96] Anna I Esparcia-Alcázar, Francisco Almenar, M Martínez, Urko Rueda, and T Vos. Q-learning strategies for action selection in the TESTAR automated testing tool. In *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*, pages 130–137, 2016.

[97] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 81–90. IEEE, 2012.

[98] Tanapuch Wanwarang, Nataniel P Borges Jr, Leon Bettscheider, and Andreas Zeller. Testing apps with real-world inputs. In *Proceedings of the 1st IEEE/ACM International Conference on Automation of Software Test (AST)*. ACM, 2020.

[99] Pedro Costa, Ana CR Paiva, and Miguel Nabuco. Pattern based GUI testing for mobile applications. In *Quality of Information and Communications Technology (QUATIC), 2014 9th International Conference on the*, pages 66–74. IEEE, 2014.

[100] Linda L Lohr. Three principles of perception for instructional interface design. *Educational Technology*, 40(1):45–52, 2000.

[101] Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro. Extracting widget descriptions from GUIs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 347–361. Springer, 2012.

[102] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data—the story so far. *International journal on semantic web and information systems*, 5(3):1–22, 2009.

[103] Amrapali Zaveri, Anisa Rula, Andrea Maurino, Ricardo Pietrobon, Jens Lehmann, and Sören Auer. Quality assessment for linked data: A survey. *Semantic Web*, 7(1):63–93, 2016.

[104] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the linked data best practices in different topical domains. In *International Semantic Web Conference*, pages 245–260. Springer, 2014.

[105] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (URI): Generic syntax. 2005.

[106] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[107] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[108] Android. Ui overview. `https://developer.android.com/guide/topics/ui/overview.html`, 2017. Accessed: 2017-10-12.

[109] Lluís Màrquez and Horacio Rodríguez. Part-of-speech tagging using decision trees. In *European Conference on Machine Learning*, pages 25–36. Springer, 1998.

[110] Kristina Toutanova and Colin Cherry. A global model for joint lemmatization and part-of-speech prediction. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 486–494. Association for Computational Linguistics, 2009.

[111] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. Insights into layout patterns of mobile user interfaces by an automatic analysis of Android apps. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 275–284. ACM, 2013.

[112] Seyed Iman Mirrezaei, Bruno Martins, and Isabel F Cruz. The triplex approach for recognizing semantic relations from noun phrases, appositions, and adjectives. In *International Semantic Web Conference*, pages 230–243. Springer, 2015.

[113] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. Mining Android app usages for generating actionable GUI-based execution scenarios. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 111–122. IEEE, 2015.

[114] Thomas F Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning design semantics for mobile apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 569–579, 2018.

[115] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53. IEEE, 2019.

[116] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. Search-based test input generation for string data types using the results of web queries. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 141–150. IEEE, 2012.

[117] Mehdi Allahyari, Seyedamin Pouriyeh, Mehdi Assefi, Saied Safaei, Elizabeth D Trippe, Juan B Gutierrez, and Krys Kochut. A brief survey of text mining: Classification, clustering and extraction techniques. *arXiv preprint arXiv:1707.02919*, 2017.

[118] Nataniel P Borges Jr, Manuel Benz, Eric Bodden, and Andreas Zeller. Testing with user interface grammars. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM, 2020. (**Manuscript under review with number TOSEM-2020-0062**).

[119] P Resnick. Rfc 2822: Internet message format, 2001, 2008.

[120] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Grammar coverage. In *Generating Software Tests*. Saarland University, 2019. Accessed: 2019-05-21.

[121] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Probabilistic grammar fuzzing. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-12-21 16:38:57+01:00.

[122] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 189–199. IEEE Press, 2019.

[123] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[124] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289*, 2018.

[125] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 238–249, 2016.

[126] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 987–992. ACM, 2016.

[127] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[128] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM, 2016.

[129] Amin Milani Fard, Mehdi Mirzaaghaei, Ali Mesbah, Amin Milani Fard, Mehdi Mirzaaghaei, Ali Mesbah, Amin Milani Fard, Mehdi Mirzaaghaei, Ali Mesbah, Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In *ASE '14 Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78, 2014.

[130] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*, pages 269–280. IEEE Press, 2019.

[131] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012.

[132] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[133] Atif M Memon, Mary Lou Soffa, and Martha E Pollack. Coverage criteria for gui testing. *ACM SIGSOFT Software Engineering Notes*, 26(5):256–267, 2001.

[134] Rahul Pandita, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Guided test generation for coverage criteria. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[135] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[136] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[137] M. Zimmermann. Travor. `https://github.com/zimmski/tavor`, 2019. Accessed: 2019-08-19.

[138] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25(1):53–95, 2013.

[139] Alexander Egyed, Gernot Binder, and Paul Grunbacher. Strada: A tool for scenario-based feature-to-code trace detection and analysis. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 41–42. IEEE Computer Society, 2007.

[140] Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 337–346. IEEE, 2005.

[141] Hira Agrawal, James L Alberi, Joseph R Horgan, J Jenny Li, Saul London, W Eric Wong, Sudipto Ghosh, and Norman Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.

[142] Orla Greevy, Stéphane Ducasse, and Tudor Girba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 347–356. IEEE, 2005.

[143] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 234–243. ACM, 2007.

[144] Nataniel P Borges Jr. Data flow oriented ui testing: exploiting data flows and ui elements to test android applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 432–435, 2017.

[145] Jie Huang, Nataniel P Borges Jr, Sven Bugiel, and Michael Backes. Up-to-crash: Evaluating third-party library updatability on android. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30. IEEE, 2019.

[146] Mohammad Naseri, Nataniel P Borges Jr, Andreas Zeller, and Romain Rouvoy. Accessileaks: Investigating privacy leaks exposed by the android accessibility service. *Proceedings on Privacy Enhancing Technologies*, 2019(2):291–305, 2019.

[147] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P Borges Jr, Eric Bodden, and Andreas Zeller. Heaps'n Leaks: How heap snapshots improve android taint analysis. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020.

[148] Nataniel P Borges Jr, Jenny Rau, and Andreas Zeller. Speeding up gui testing by on-device test generation. In *Proceedings of the 35rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2020.

Appendices

Appendix **A**

Apps and Results for DM-2

Table 13: Apps used to evaluate DM-2

App	**Source**	**Downloads**	**Category**
Alogblog	F-Droid	-	Internet
KeePassDroid (2.0.6.4)	F-Droid	-	Security
BART Runner (2.2.6)	F-Droid	-	Navigation
Jamendo (1.0.4)	F-Droid	-	Music
DroidWeight (1.3.3)	F-Droid	-	Sports & Health
Pizza Cost (1.05-9)	F-Droid	-	Money
Munch (0.44)	F-Droid	-	Internet
Mirrored (0.2.9)	F-Droid	-	Internet
World Weather (1.2.4)	Play Store	1k-5k	Weather
SyncMyPix (0.16)	Play Store	250k-500k	Social
Der Die Das (16.04.2016)	Play Store	500k-1M	Learning
wikiHow (2.7.3)	Play Store	1M-5M	Books

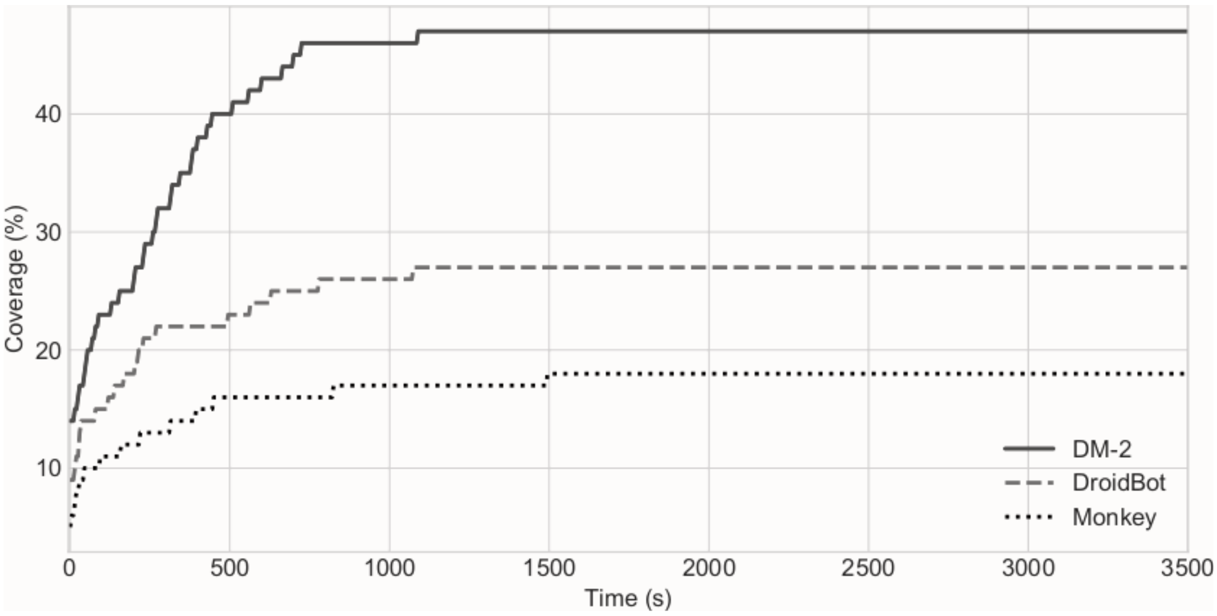


Figure 51: Coverage over time between DM-2, DroidBot and Monkey for Alogblog app (a2dp.Vol)

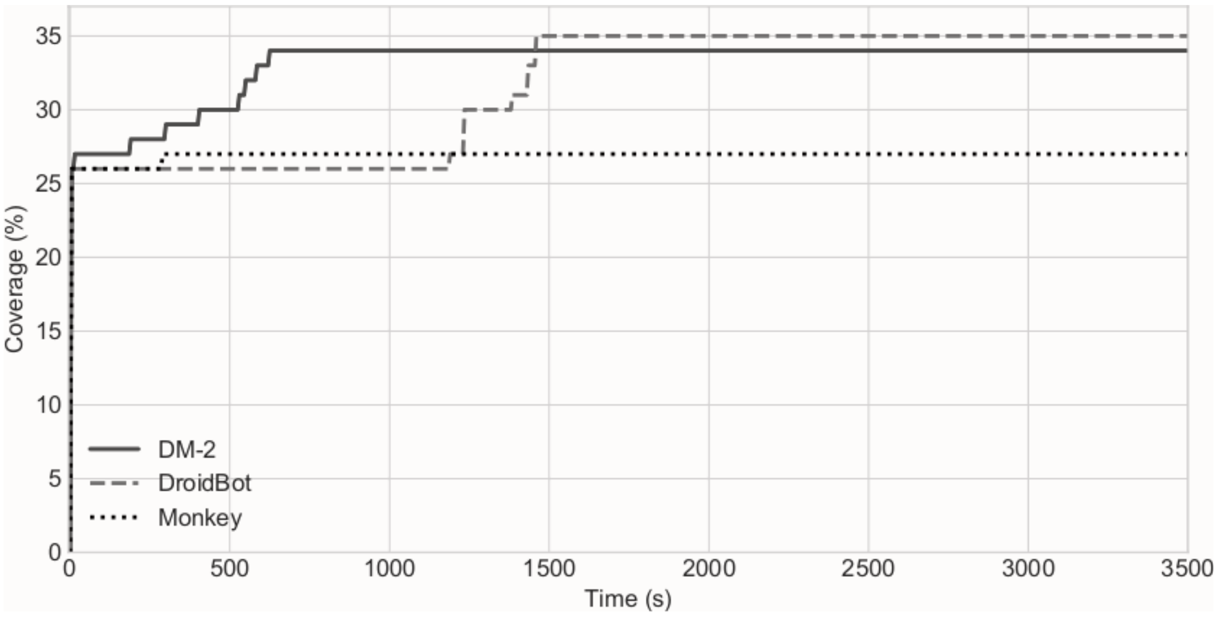


Figure 52: Coverage over time between DM-2, DroidBot and Monkey for KeePassDroid app (com.android.keepass)

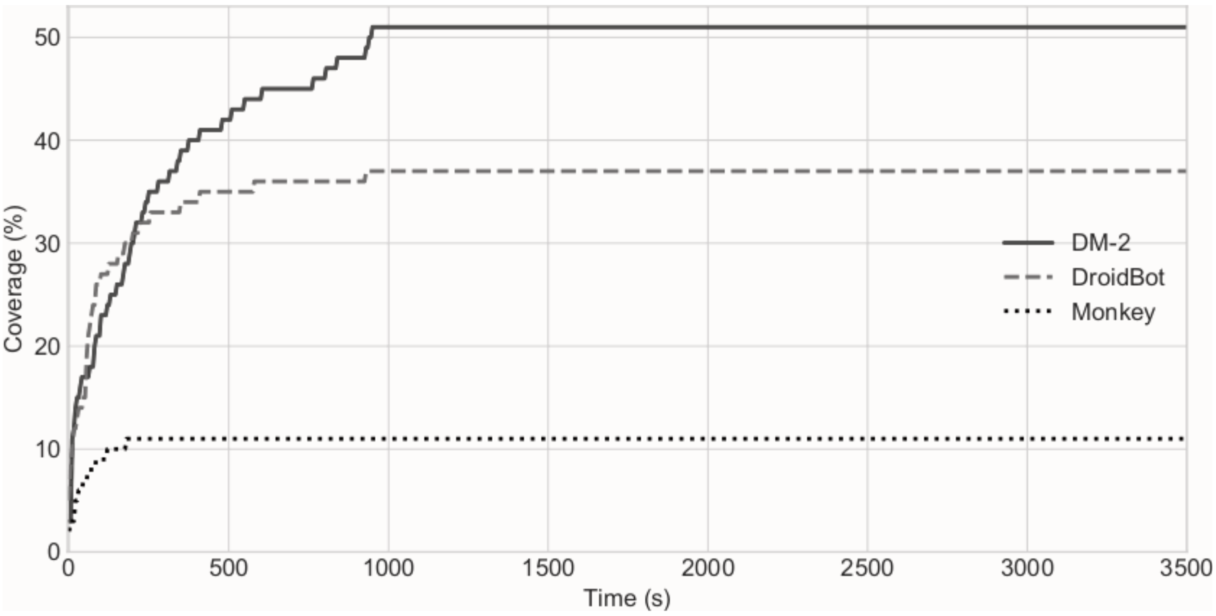


Figure 53: Coverage over time between DM-2, DroidBot and Monkey for Munch app (com.crazyhitty.chdev.ks.munch)

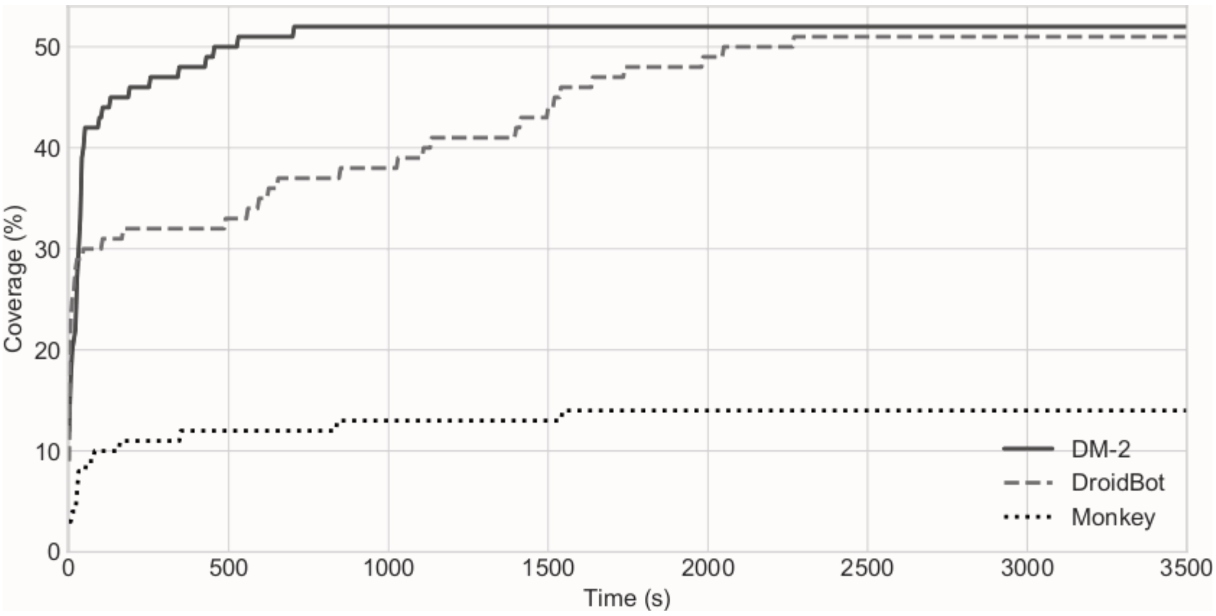


Figure 54: Coverage over time between DM-2, DroidBot and Monkey for BART Runner app (com.dougkeen.bart)

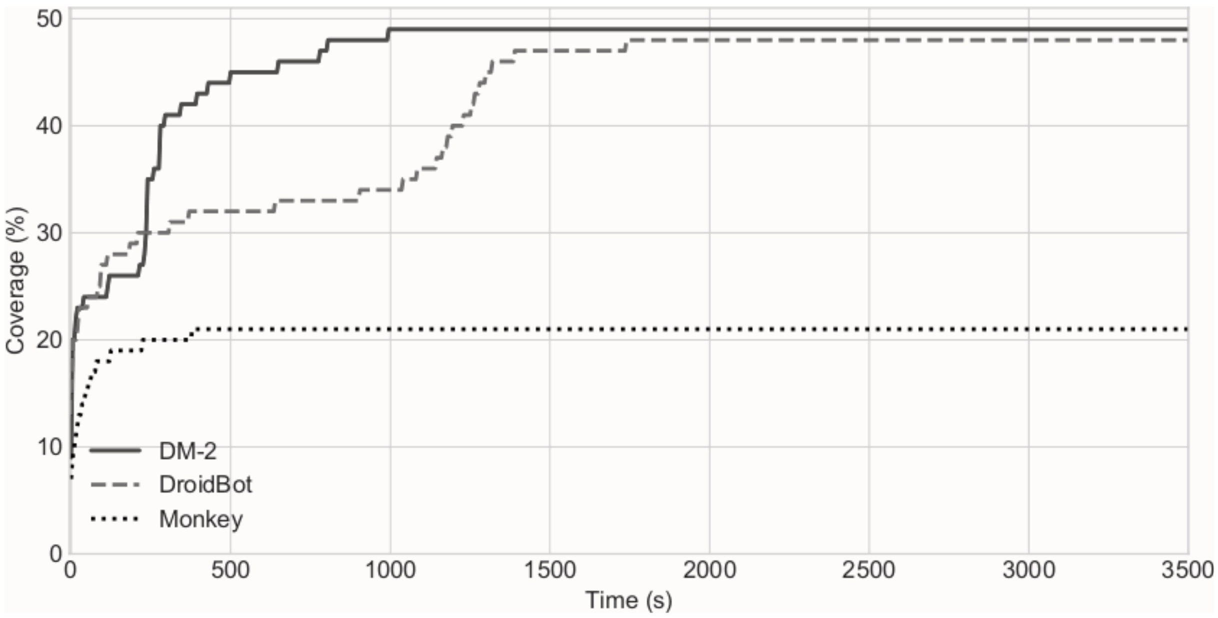


Figure 55: Coverage over time between DM-2, DroidBot and Monkey for World Weather app (com.haringeymobile.ukweather)

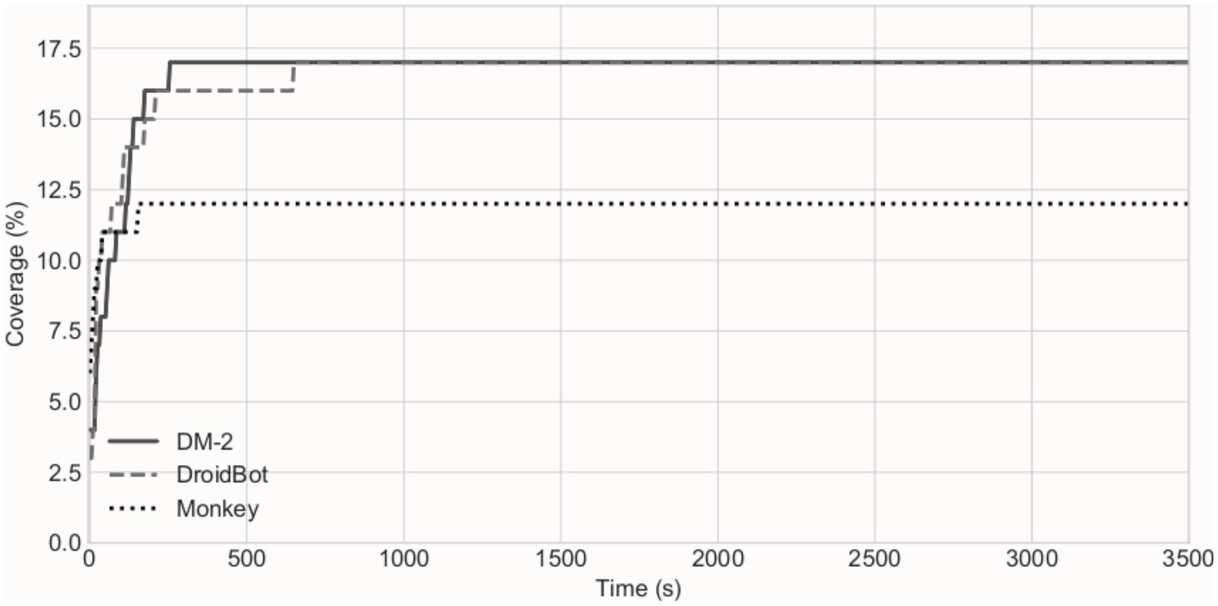


Figure 56: Coverage over time between DM-2, DroidBot and Monkey for SyncMyPix app (com.nloko.android.syncmypix)

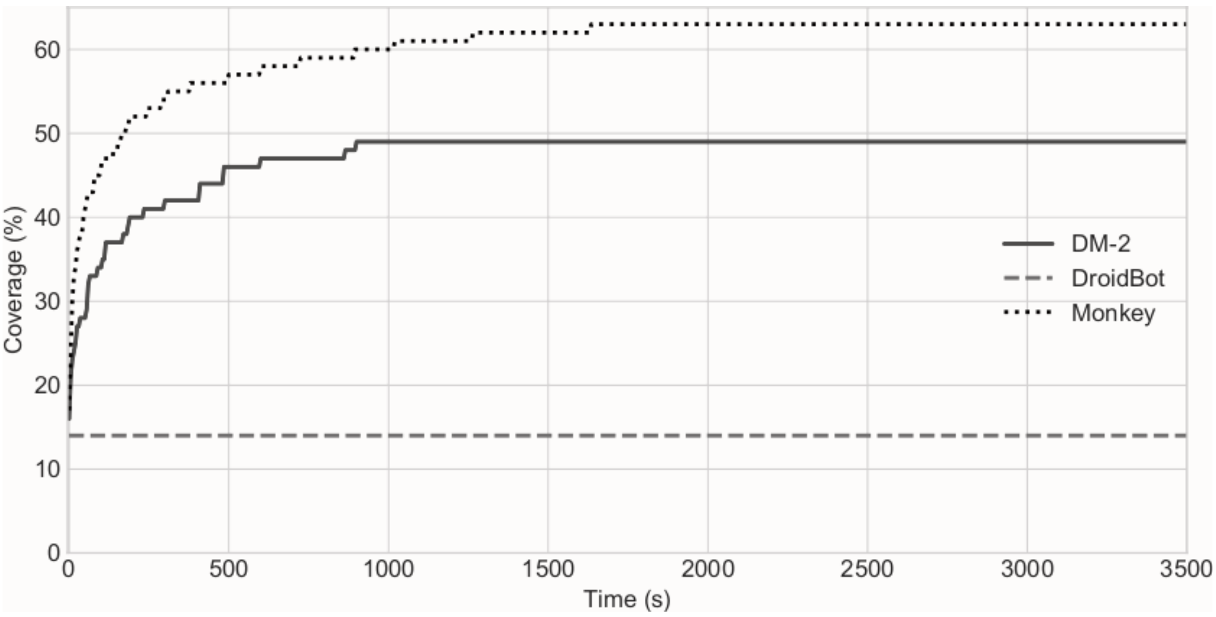


Figure 57: Coverage over time between DM-2, DroidBot and Monkey for Der Die Das app (com.lubosmikusiak.articuli.derdiedas)

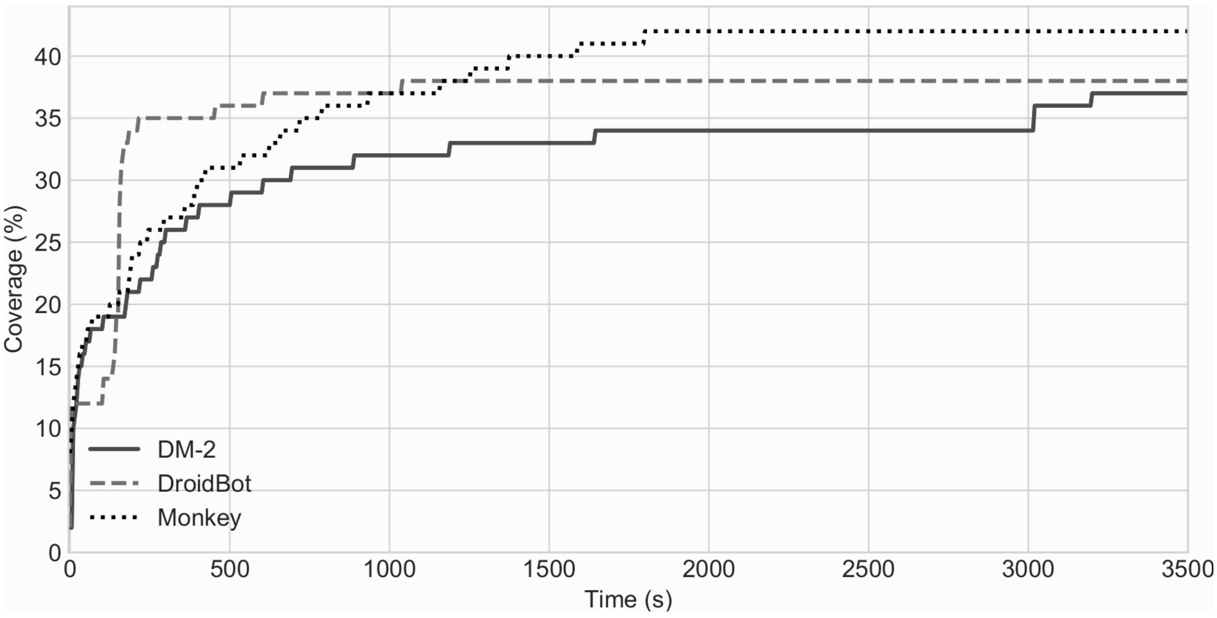


Figure 58: Coverage over time between DM-2, DroidBot and Monkey for Jamendo app (com.teleca.jamendo)

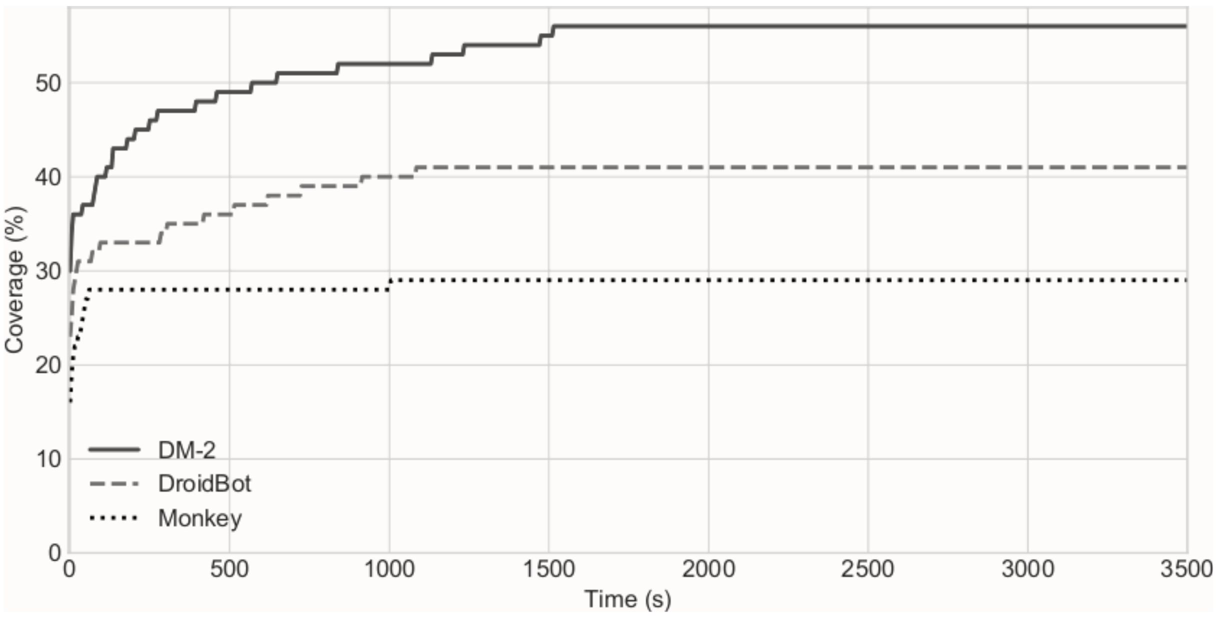


Figure 59: Coverage over time between DM-2, DroidBot and Monkey for WikiHow app (com.wikihow.wikihowapp)

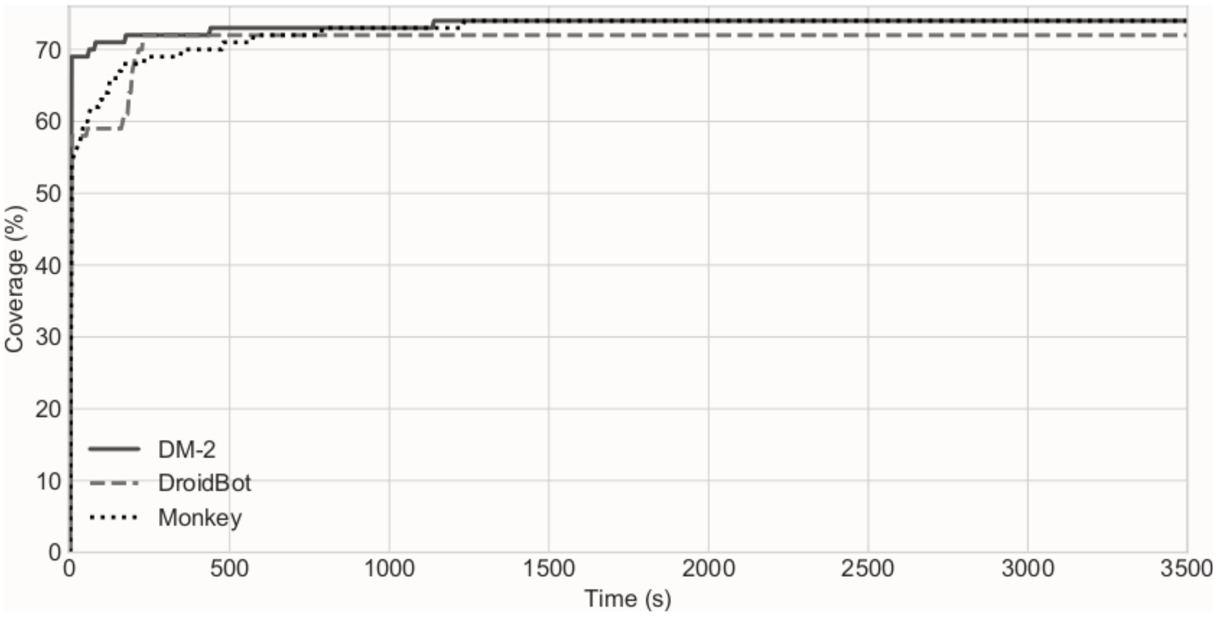


Figure 60: Coverage over time between DM-2, DroidBot and Monkey for Pizza Cost app (de.drhoffmannsoft.pizza)

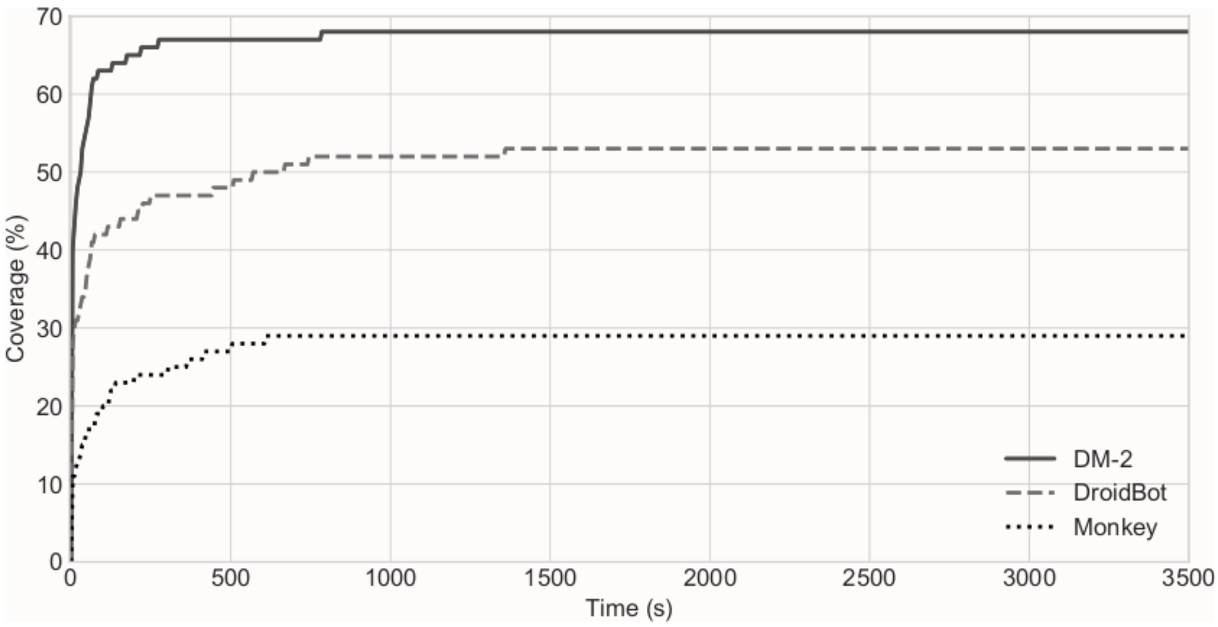


Figure 61: Coverage over time between DM-2, DroidBot and Monkey for Mirrored app (de.homac.Mirrored)

Appendix B

Apps and Results for Static UI Interaction Model Experiments

Table 14: Apps used to evaluate the *UI interaction model*

App	Source	Downloads	Category
Alogblog	F-Droid	-	Internet
KeePassDroid (2.0.6.4)	F-Droid	-	Security
Munch (0.44)	F-Droid	-	Internet
BART Runner (2.2.6)	F-Droid	-	Navigation
Jamendo (1.0.4)[15]	F-Droid	-	Music
2048 (2.06)	F-Droid	-	Games
DroidWeight (1.3.3)	F-Droid	-	Sports & Health
Pizza Cost (1.05-9)[15]	F-Droid	-	Money
Mirrored (0.2.9)	F-Droid	-	Internet
Easy xkcd (5.3.9)	F-Droid	-	Internet
Dialer2 (2.90)	F-Droid	-	Phone & SMS
PasswordMaker (1.1.11)	F-Droid	-	Security
Tomdroid (0.4.1)	F-Droid	-	Writing
World Weather (1.2.4)	Play Store	1k-5k	Weather
SyncMyPix (0.16)	Play Store	250k-500k	Social
Der Die Das (16.04.2016)	Play Store	500k-1M	Learning
wikiHow (2.7.3)	Play Store	1M-5M	Books

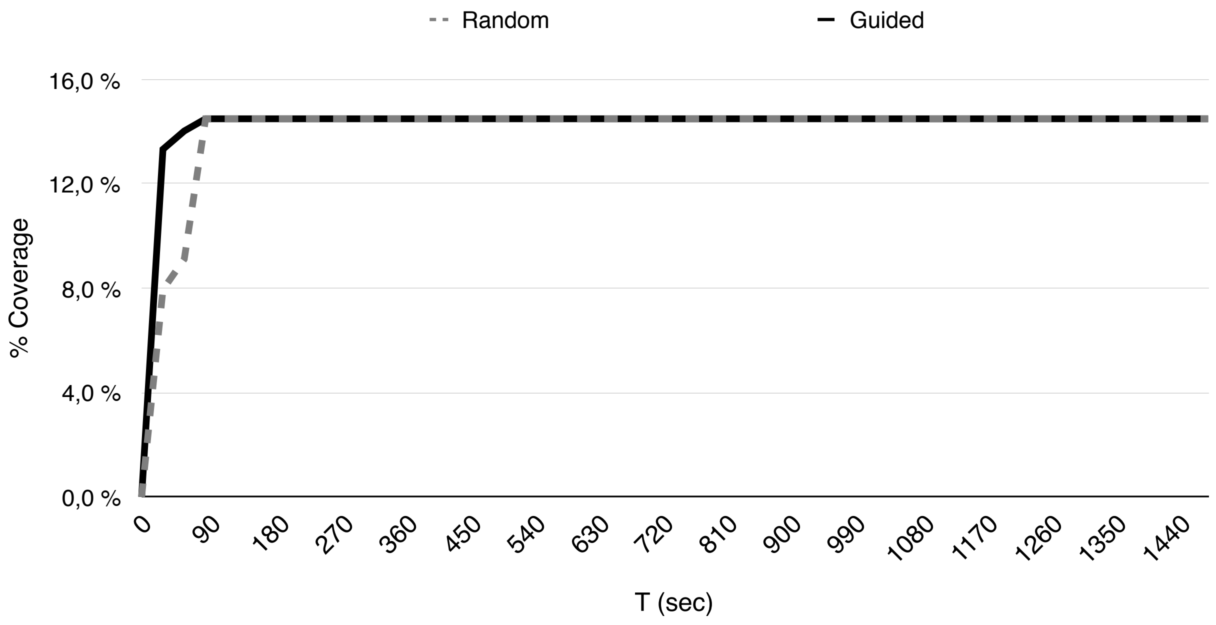


Figure 62: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for Alogblog app (com.alogblog.aaa)

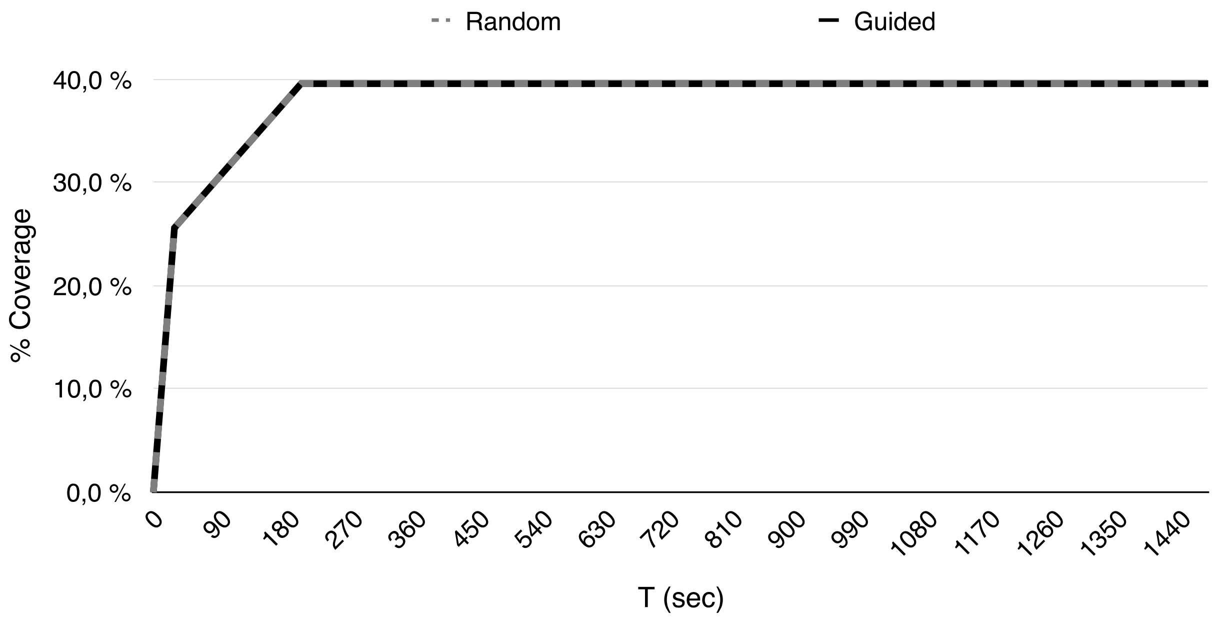


Figure 63: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for KeppPassDroid app (com.android.keepass)

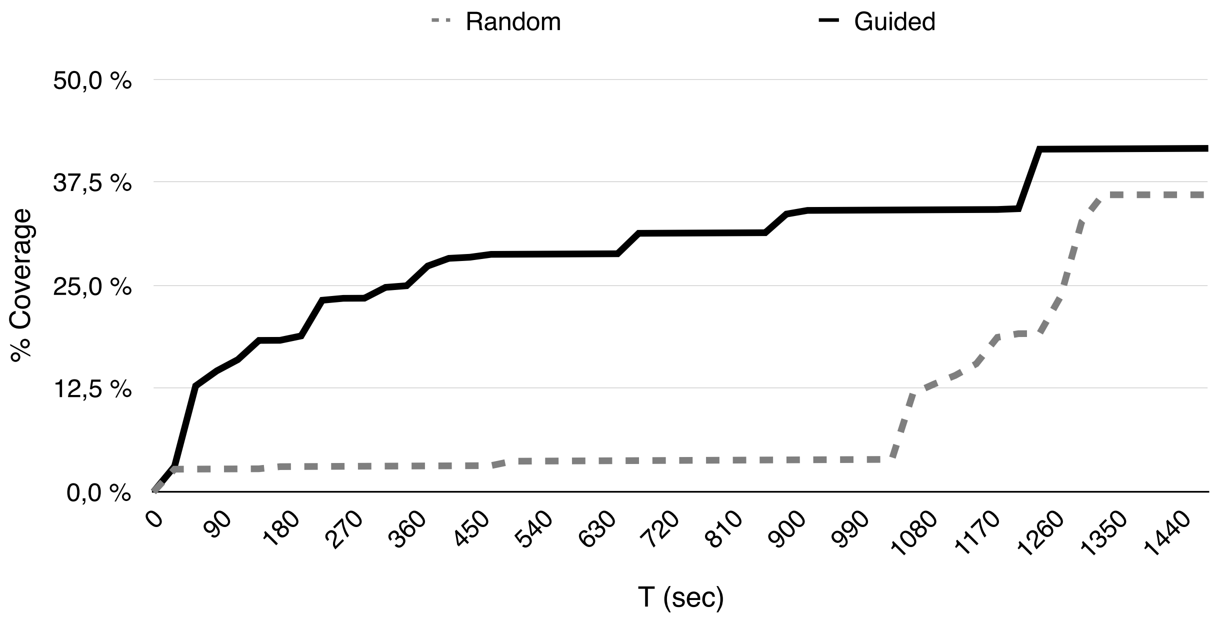


Figure 64: Coverage over time between DM-2 (Random) and DROIDMATE-M (Guided) for Munch app (com.crazyhitty.chdev.ks.munch)

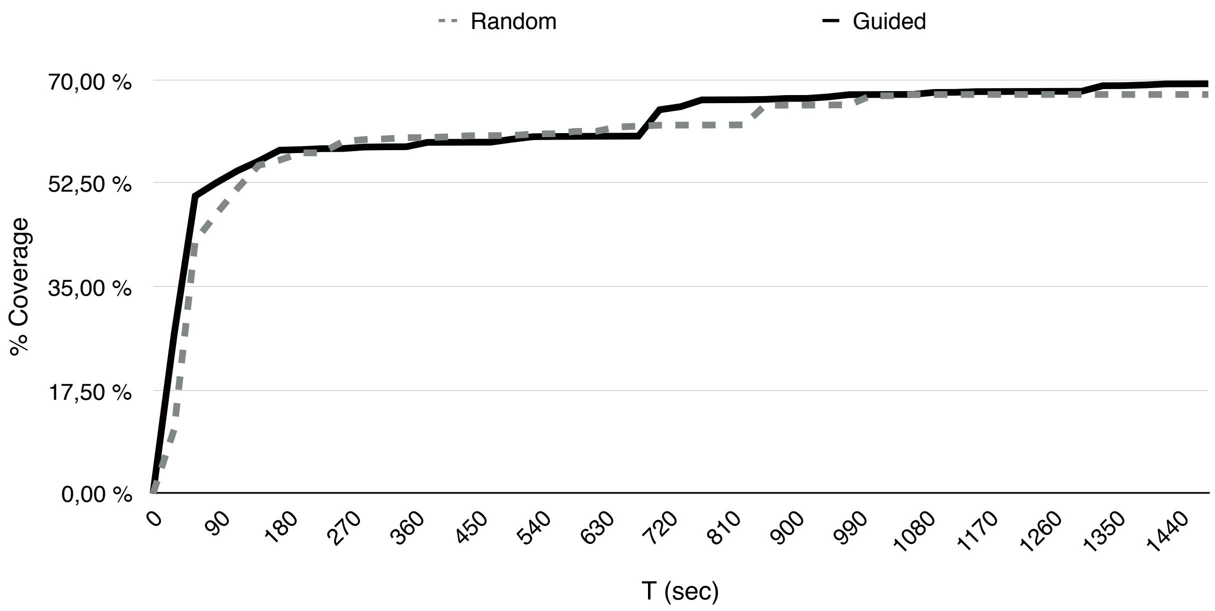


Figure 65: Coverage over time between DM-2 (Random) and DROIDMATE-M (Guided) for BART Runner app (com.dougkeen.bart)

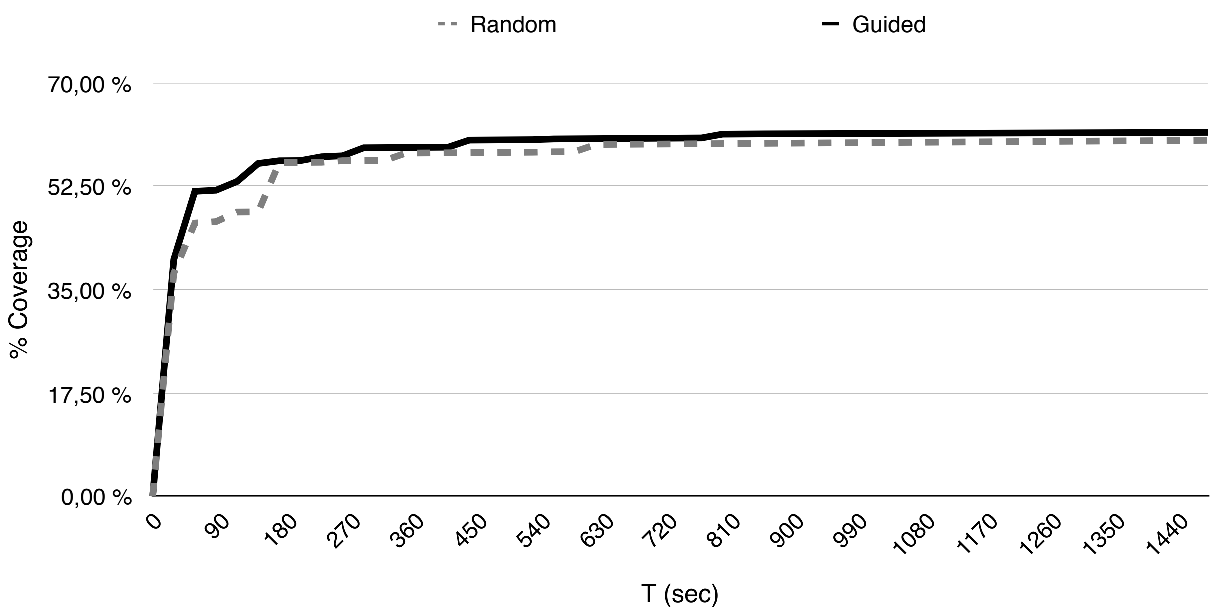


Figure 66: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for World Weather app (com.haringeymobile.ukweather)

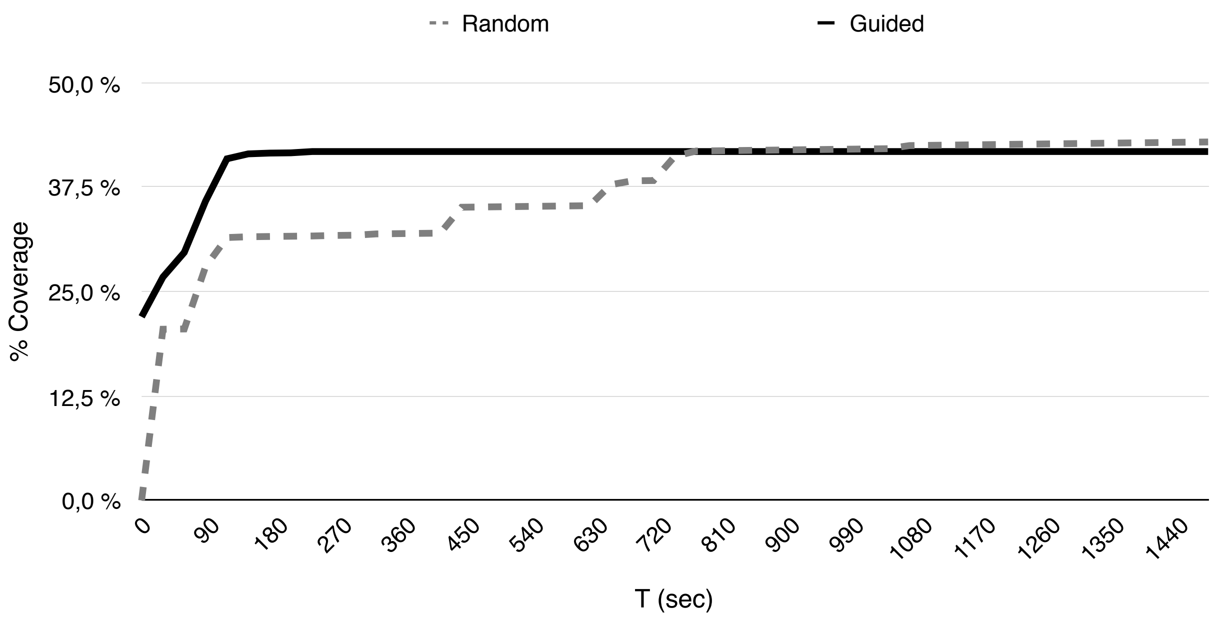


Figure 67: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for Der Die Das app com.lubosmikusiak.articuli.derdiedas

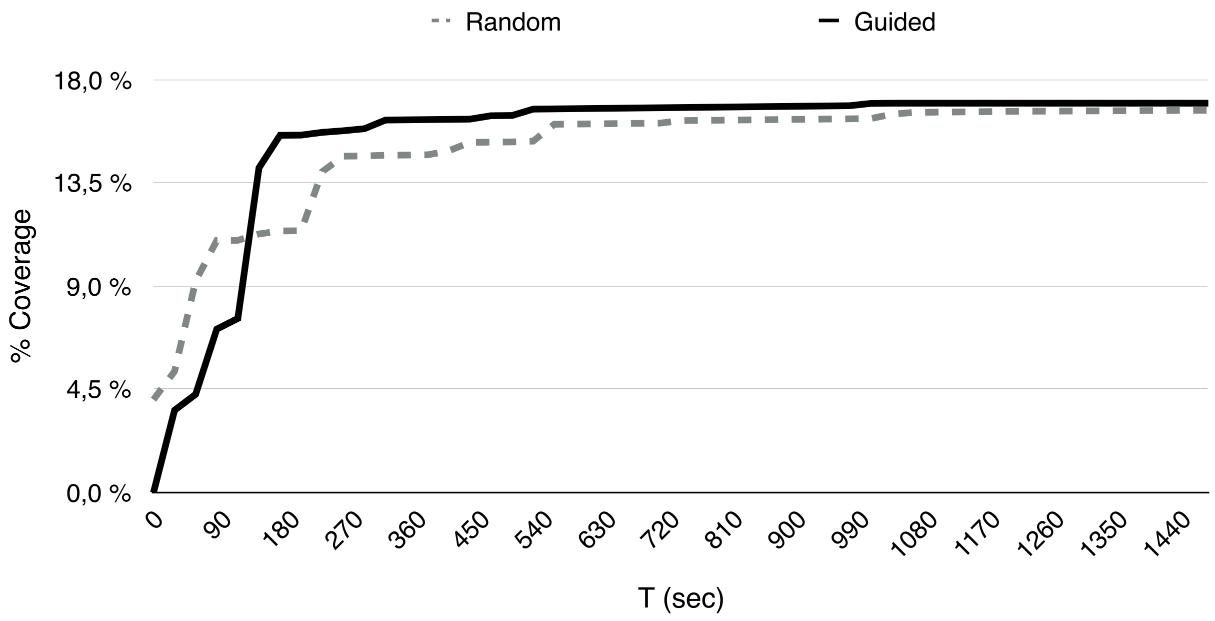


Figure 68: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for SyncMyPix app (com.nloko.android.syncmypix)

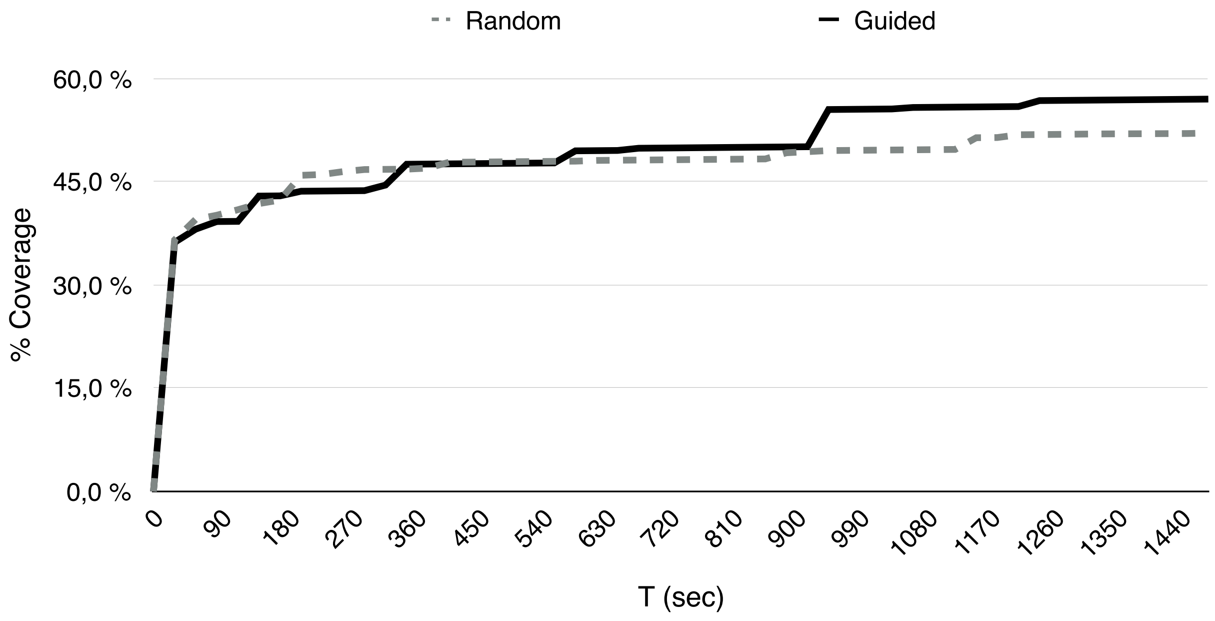


Figure 69: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for WikioHow app (com.wikihow.wikihowapp)

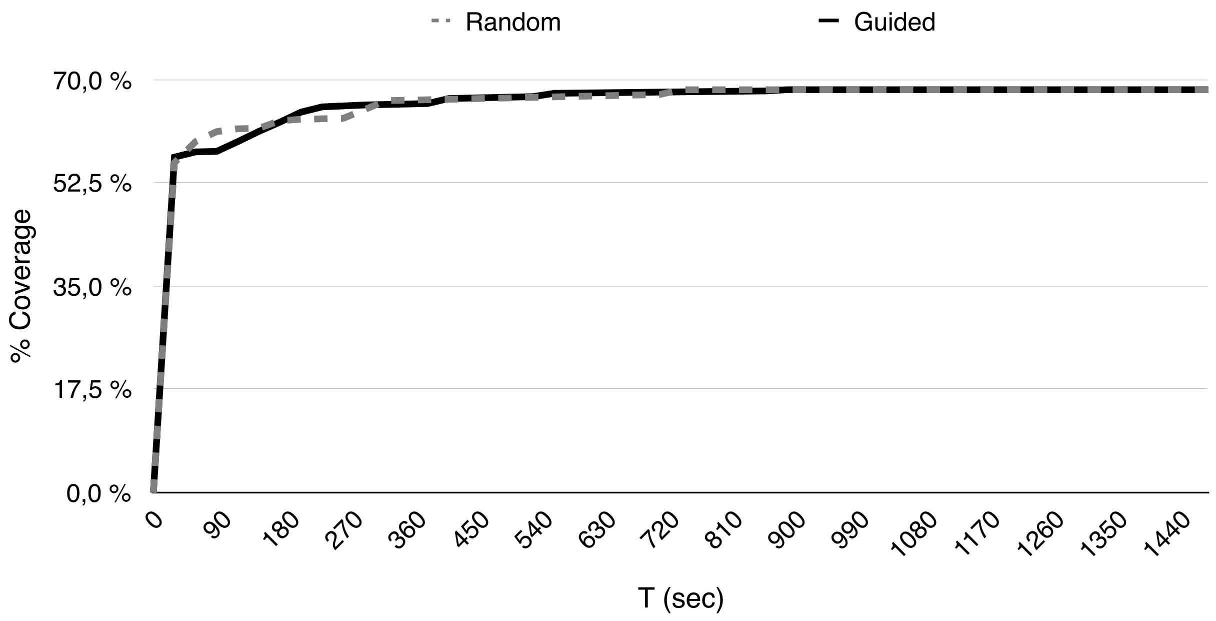


Figure 70: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for Pizza Cost app (de.drhoffmannsoft.pizza)

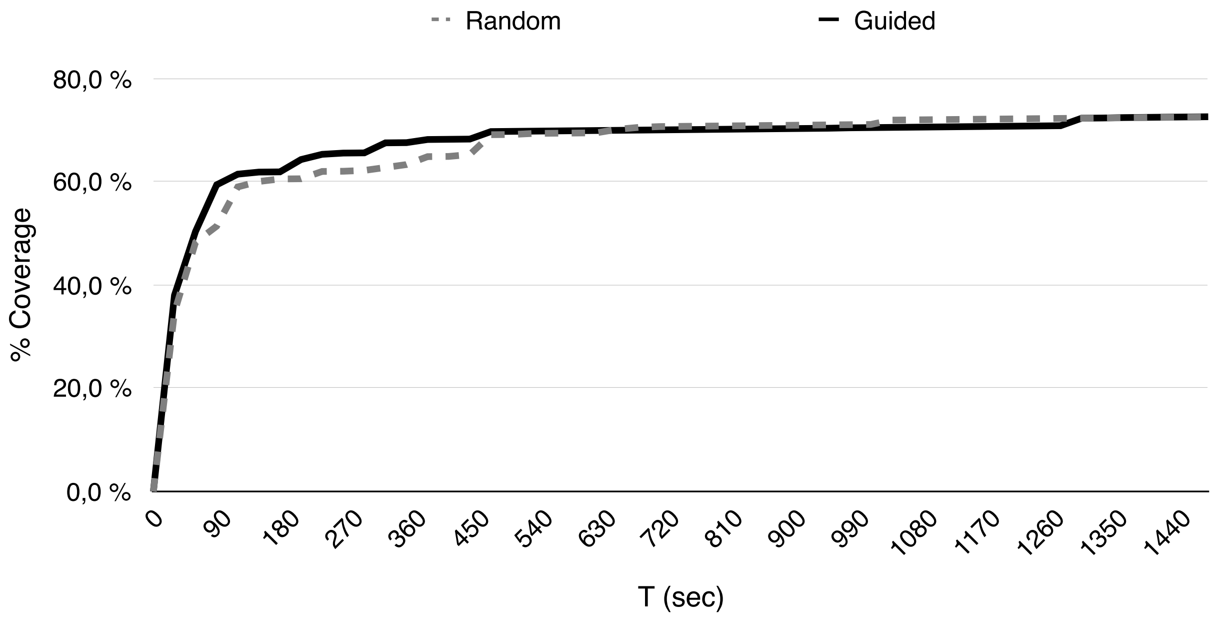


Figure 71: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for Mirrored app (de.homac.Mirrored)

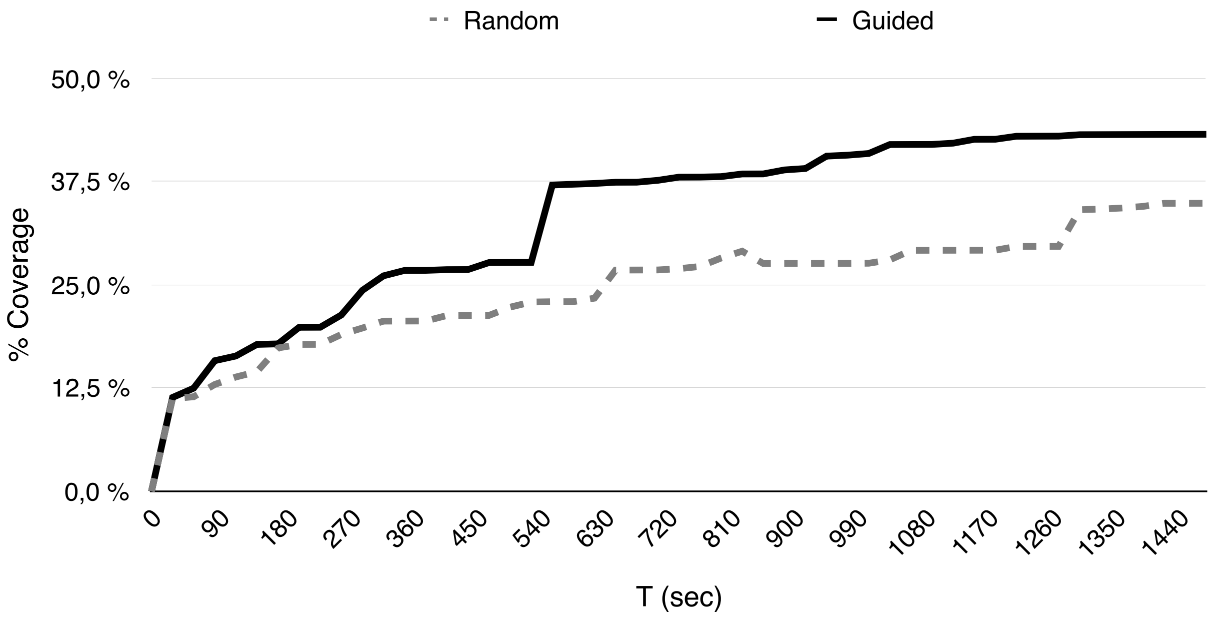


Figure 72: Coverage over time between DM-2 (Random) and DROIDMATE-M (Guided) for Easy XKCD app (de.tap.easy.xkcd)

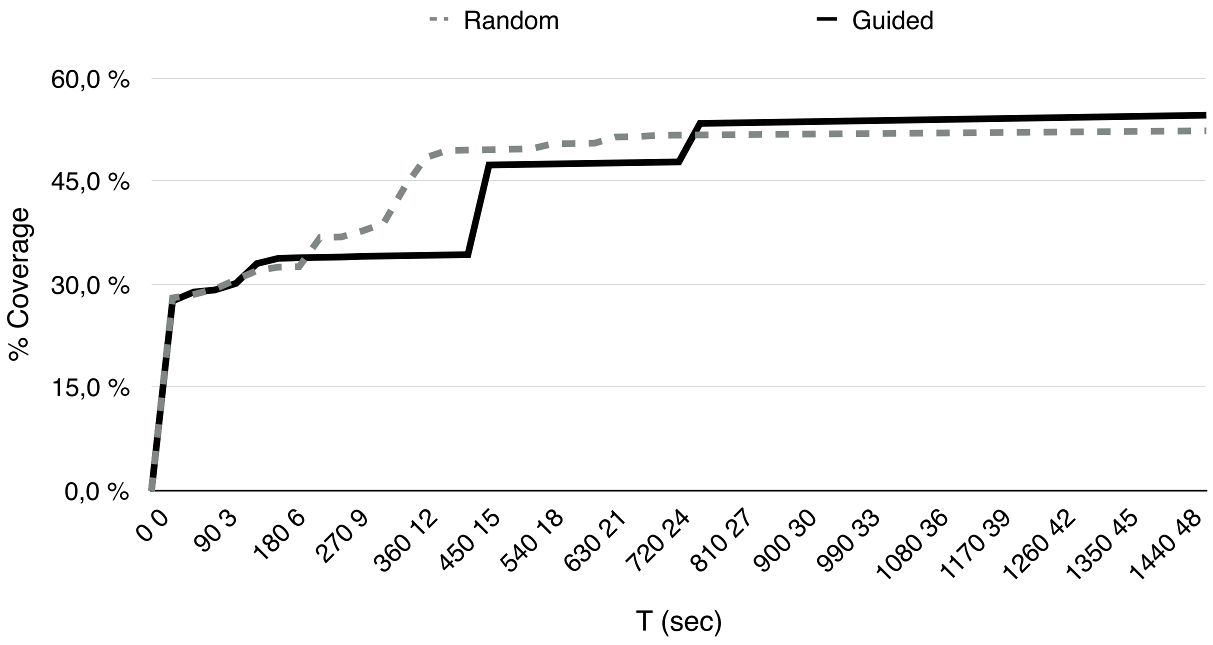


Figure 73: Coverage over time between DM-2 (Random) and DROIDMATE-M (Guided) for Dialer2 app (org.dnaq.dialer2)

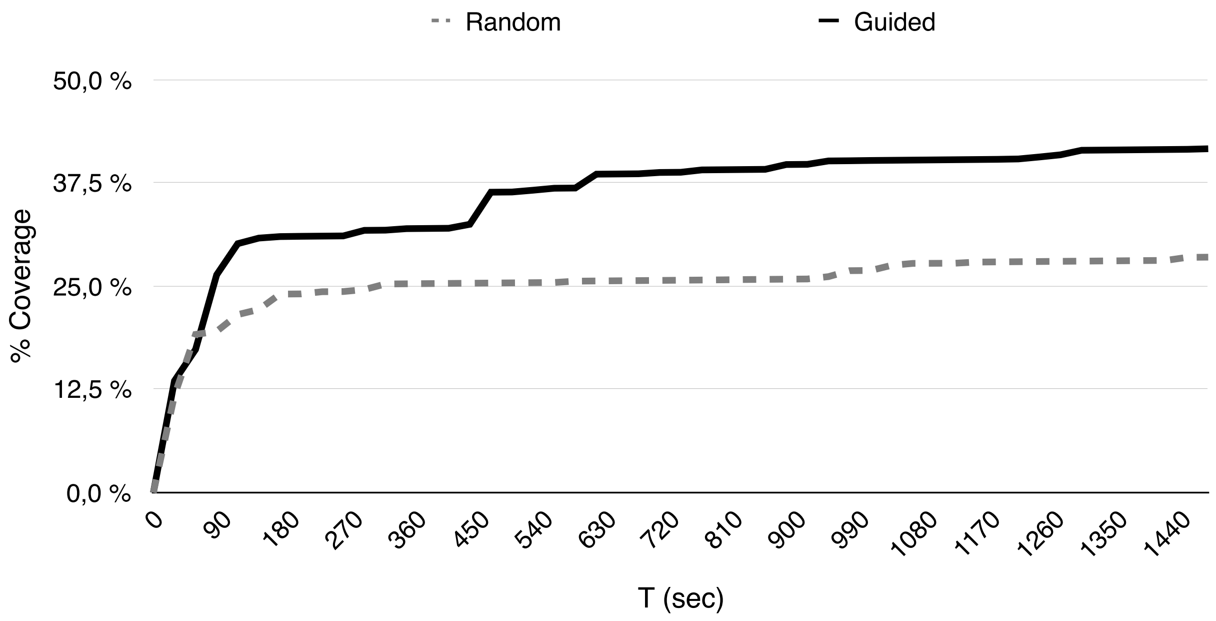


Figure 74: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for PasswordMaker app (org.passwordmaker.android)

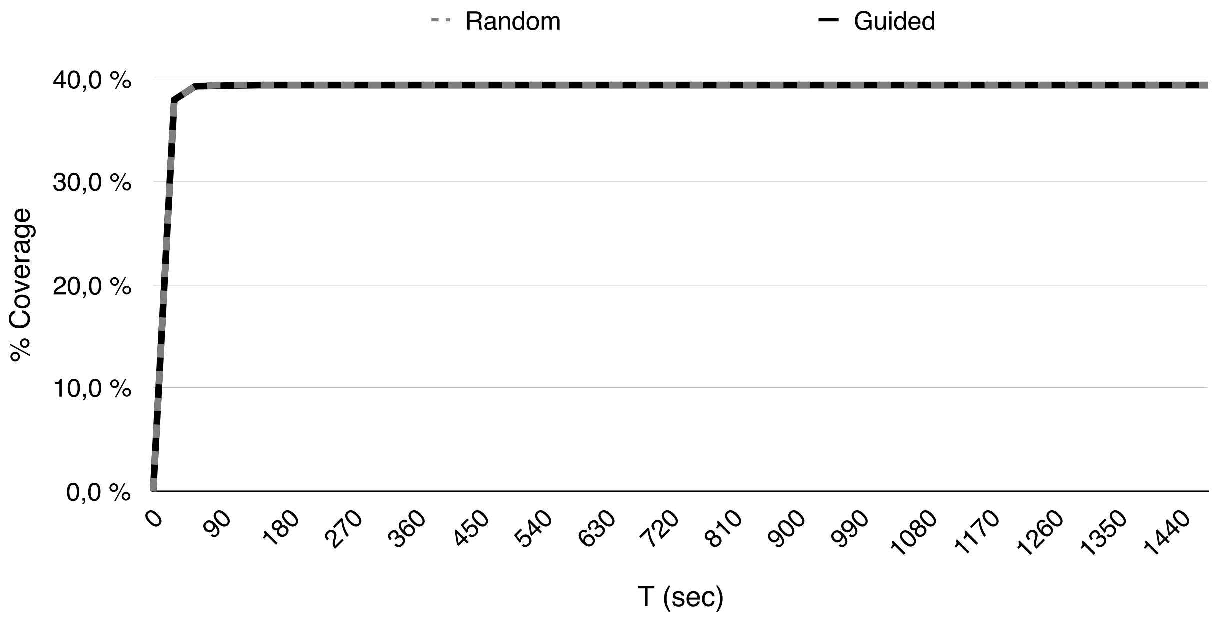


Figure 75: Coverage over time between DM-2 (Random) and Droidmate-M (Guided) for TomDroid app (org.tomdroid)

Appendix C

Apps and Results for Reinforcement Learning Experiments

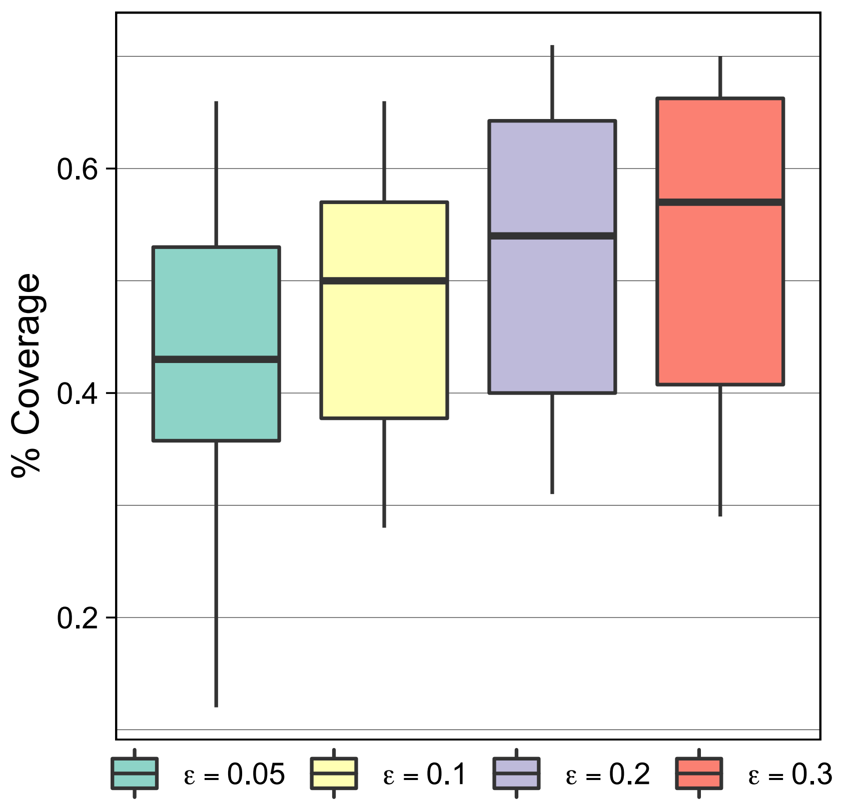


Figure 76: Results for the parameter tuning experiments for selecting ϵ for ϵ-Greedy strategy

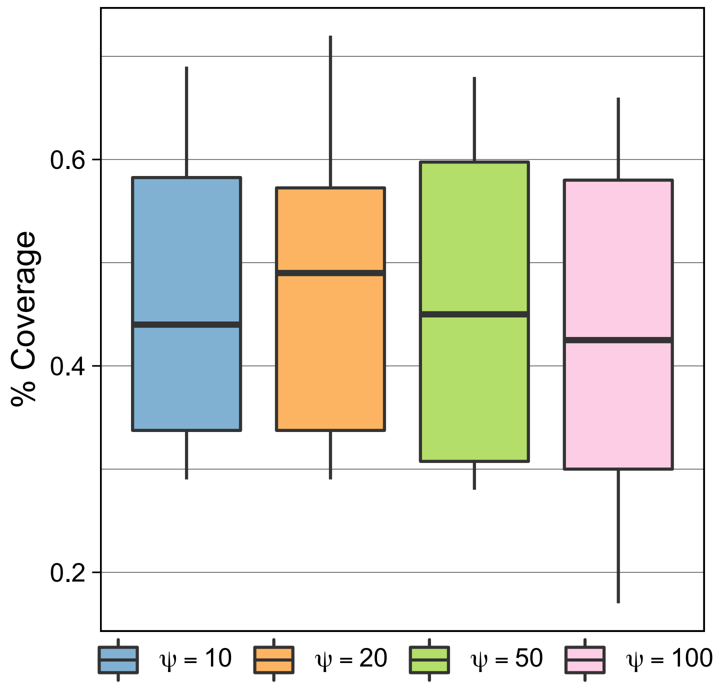


Figure 77: Results for the parameter tuning experiments for selecting the weight of the static model (ψ)

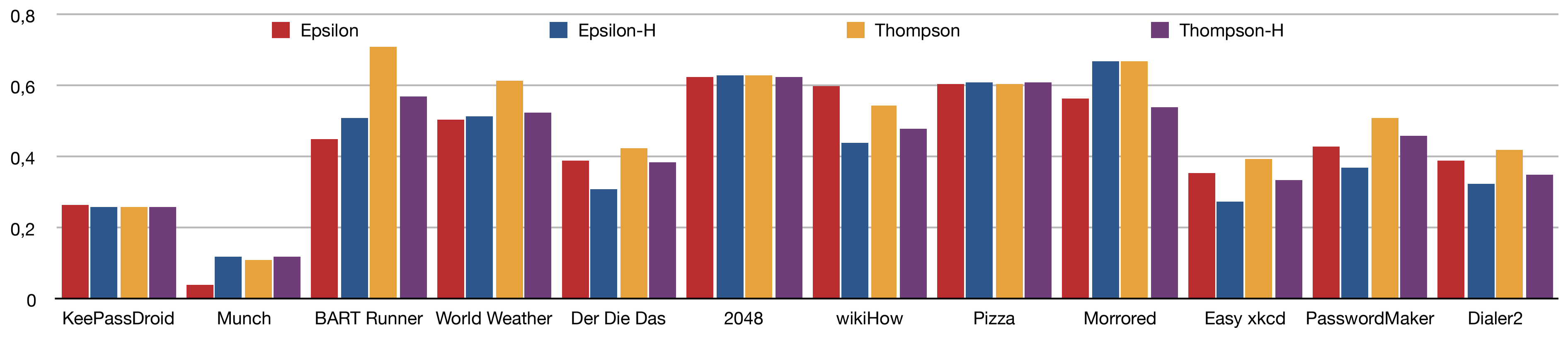


Figure 78: Per app comparison of statement coverage between ϵ-Greedy, ϵ-Greedy+K, Thompson, and Thompson+K

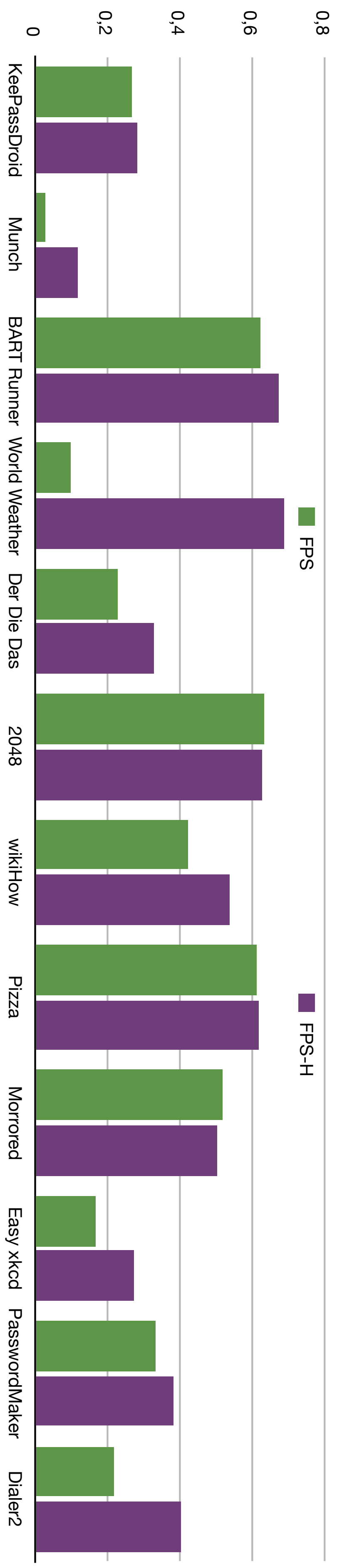


Figure 79: Per app comparison of statement coverage between Baseline, and Baseline+K

Appendix D

Apps and Results for UI and Code Grammars

Table 15: Apps used in the experiments and the coverage achieved by the initial test case

Name	**Version**	**Statements**	**Test Coverage (%)**
AAT	1.8	31124	12201 (39%
ALSA Mixer WebUI	0.3.2	2633	504 (19%)
Altcoin Prices	1.7.0	6191	2913 (47%)
BART Runner	2.2.6	8095	3934 (49%)
BeHe Pro	2.6.4	4524	2646 (58%)
Bits & Baeume	1.37.3	4434	1520 (34%)
Boilr	0.7.0	10955	3248 (30%)
BookList	1.7	1438	231 (16%)
Car Report	3.25.0	17417	2736 (16%)
Conversations Legacy	2.5.8	64624	7152 (11%)
Cool Mic	1.0.6	2750	1769 (64%)
Cool Reader	3.2.9-1	53883	11118 (21%)
CPU Info	4.2.0	14283	7018 (49%)
Crates.io unofficial	1.3.2	3478	1270 (37%)
Democracy Droid	3.7.1	3544	1803 (51%)
Diceware Password Generator	1.8	1873	568 (30%)
Easy Diary	1.4.1973	14044	5783 (41%)
Easy Weather	1.1	1763	392 (22%)
eBooks	0.4	1375	296 (22%)
Episodes	0.12	4441	840 (19%)
F-Droid	1.7	41409	14067 (34%)
Fake Traveler	1.6	1347	270 (20%)
Fancy Places	1.2.4	3731	1878 (50%)
Flite TTS Engine	3.0.0	1493	981 (66%)
FOSDEM Companion	1.6.2	11843	6719 (57%)
FOSSASIA	1.0.1	13983	6015 (43%)
Goblim	2.8	3725	955 (26%)
Good Weather	4.4	5301	3122 (59%)
Halachic Times	6.0	12444	6086 (49%)
Inventum	0.3	6813	4490 (66%)
Just Craigslist	2.0	6738	4181 (62%)
KouChat	1.1.1	12153	3809 (31%)
LinuxDayOSM	1.6.2	2972	1587 (53%)
My Expenses	3.0.0	72056	15970 (22%)
MyHackerspace	1.8.2	2260	1189 (53%)
NewPipe	0.16.1	49460	16095 (33%)
NWS Weather Alerts Widget	1.1.3	2647	1345 (51%)
RedReader	1.9.10	43357	14462 (33%)
Simple Last.fm Scrobbler	1.5.7	12279	3422 (28%)
Single-Feed	2.8	2276	1248 (55%)
SmartNavi	2.2.4	6467	2621 (41%)
taz.app	3.9.2.3	26134	11199 (43%)
Transportr	2.0.4	21124	4984 (24%)
Tricky Tripper	1.6.0	16677	4590 (28%)
Vlille Checker	4.2.3	3552	2260 (64%)
Your forecast	4.7.7	26781	10014 (37%)

Table 16: Size of the mined grammar for individual apps

Name	Prod. Rules	State Prod.	Widgets
AAT	107	66	36
ALSA Mixer WebUI	10	403	367
Altcoin Prices	236	41	42
BART Runner	203	148	132
BeHe Pro	619	65	81
Bits & Baeume	20	39	25
Boilr	140	10	30
BookList	17	20	42
Car Report	302	259	301
Conversations Legacy	189	157	102
Cool Mic	60	124	171
Cool Reader	95	80	57
CPU Info	31	5	13
Crates.io unofficial	104	452	15
Democracy Droid	138	36	25
Diceware Password Generator	23	2	2
Easy Diary	206	71	34
Easy Weather	16	126	43
eBooks	94	80	62
Episodes	30	121	67
F-Droid	114	79	74
Fake Traveler	199	19	87
Fancy Places	21	33	40
Flite TTS Engine	94	57	43
FOSDEM Companion	238	33	12
FOSSASIA	120	117	44
Goblim	86	65	43
Good Weather	80	146	129
Halachic Times	95	6	11
Inventum	167	259	206
Just Craigslist	95	68	36
KouChat	334	138	105
LinuxDayOSM	17	136	112
My Expenses	514	148	132
MyHackerspace	103	73	78
NewPipe	178	88	120
NWS Weather Alerts Widget	45	6	9
RedReader	331	11	23
Simple Last.fm Scrobbler	141	4	16
Single-Feed	75	6	8
SmartNavi	148	224	51
taz.app	135	53	97
Transportr	79	16	20
Tricky Tripper	215	25	37
Vlille Checker	56	118	59
Your forecast	186	76	59

List of Figures

List of Tables

List of Algorithms