Guiding App Testing with Mined Interaction Models

Nataniel P. Borges Jr.
Saarland University
Saarbrücken, Germany
nataniel.borges@cispa.saarland

Maria Gómez
Saarland University
Saarbrücken, Germany
maria.gomez@uni-saarland.de

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

ABSTRACT

Test generators for graphical user interfaces must constantly choose which UI element to interact with, and how. We *guide* this choice by *mining* associations between UI elements and their interactions from the most common applications. Once mined, the resulting *UI interaction model* can be easily applied to new apps and new test generators. In our experiments, the mined interaction models lead to code coverage improvements of 19.41% and 43.03% on average on two state-of-the-art tools (DROIDMATE and DROIDBOT), when executing the same number of actions.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Automated static analysis*; *Dynamic analysis*;

KEYWORDS

mobile testing, model mining, Android

ACM Reference Format:
Nataniel P. Borges Jr., Maria Gómez, and Andreas Zeller. 2018. Guiding App Testing with Mined Interaction Models. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3197231.3197243

1 INTRODUCTION

Mobile applications face a large competition. Google Play Store, the official market for Android apps, now holds more than 3.3 million apps [29]. To survive in this competition, developers need to ensure a high quality of their apps; and testing is one of the central technique to achieve this quality. It is thus no surprise that recent testing research has focused on mobile apps, notably on *test generators* [14] that synthesize inputs in order to cover behavior as effectively and efficiently as possible.

One of the central problems in generating input for graphical user interfaces is that most elements visible on a user interface are *passive,* that is, they do not respond to interactions at all. Other elements expect specific interactions, such as clicks, swipes, or test inputs. For humans, it is typically easy to identify active elements, since user interfaces follow *conventions* that humans would learn over time. As a simple example, consider the app screen shown in Figure 1, showing the change log in the popular Android app *2048 Puzzle Game*[1]. On this screen, one may scroll through the list of changes by swiping up and down, and eventually dismiss the screen by pressing the OK button at the bottom. All this is straightforward for any user with a minimal experience of mobile apps.

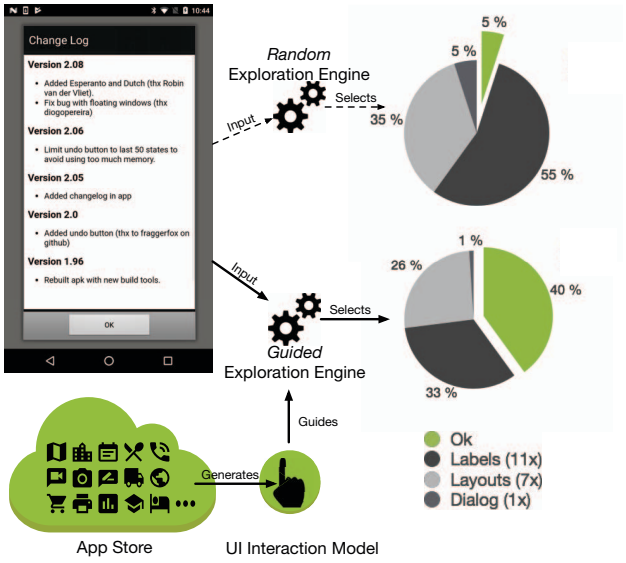


Figure 1: Guiding Test Generation with Mined Interaction Models. Traditional GUI testing (top) randomly distributes generated events across all visible UI elements, leaving only a 5% chance to click on the single active OK button. Our approach (bottom) learns from existing apps which UI elements typically 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.

For a test generator, though, nothing of this is obvious. The individual changes might respond to clicks as well; the list might also 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. From the standpoint of a test generator, all of these interactions are equally likely to result in the coverage of additional 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

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

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3197243

runtime, and not even use standard UI elements: The *2048* change log shown in Figure 1 is actually *Web content,* which may tie in JavaScript functions for further interaction both with the Web and the app, posing serious problems for analysis. In all generality, determining whether and how an app responds to an interaction is an instance of the halting problem.

In this paper, we hence follow a simpler, 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. During dynamic exploration of apps, when a UI element is found, the UI interaction model predicts with a certain probability, if the UI element will 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 1. Here, we find that the individual change log entries are *labels,* which are unlikely to respond to clicks; whereas the OK button is a *button,* which typically *does* respond to clicks. By guiding the previously random exploration towards the (likely) button and away from the (unlikely) labels, we can increase the chances of the OK button being clicked eightfold, or to 40%. Our approach is easy to implement: The UI interaction model is mined once from the most common apps, and can be reused again and again with different apps and testing tools. In our experiments, extending existing test generators to leverage the model took less than three hours of work.

In the remainder of this paper, after introducing details of the Android UI (Section 2), we make two contributions:

(1) We introduce our approach of mining a *UI Interaction Model* which captures normal behavior of UIs in Android apps (Section 3).
(2) We show how to *extend existing test generators using such mined knowledge*, guiding test generation towards likely successful UI elements and interactions (Section 4). Specifically, we extend two state-of-the-art testing tools: Droidmate and Droidbot.

Leveraging mined interaction models is efficient and effective: In our evaluation (Section 5), we find that the crowd-based approach improves the coverage of explorations by almost 20%, on average. We thus contribute to *test generation*, since interaction models improve code and UI coverage significantly. After discussing related work (Section 7), Section 8 concludes the paper and outlines future work.

2 BACKGROUND: ANDROID UI

Android apps are driven by *events*. These come in the form of *user interaction events*, such as screen touches or button presses; as well as *system events*, such as messages received or incoming calls. In this paper, we focus on user interaction events.

In Android apps, the user interface is defined as a hierarchy of *View* and *ViewGroup* objects. A *View* denotes any element displayed on the screen and users can interact with. *ViewGroups* are structuring elements which aggregate *Views* and other *ViewGroups* [5].

To intercept user interaction events, Android provides a number of *event listener interfaces* that declare public event handler methods. The Android framework invokes these event handler methods when the event occurs [5]. For example, when a *View* (such as a Button) is clicked, the *onClickEvent* method on the respective object is invoked.

While it is technically possible to attach event handlers to any *View* or *ViewGroup*, 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. For example, *TextViews* and *Layouts* typically display or structure information, and seldom have events attached.

Android natively provides tools to capture the current UI state of a device. These tools, however, make available a limited amount of information. Basically, this information reduces to the current UI's hierarchy, i.e., which views exist on the UI and how they are structured (parent-children relationship). For each view, it provides the content (textual and graphical) and state (visible, enabled and clickable). The associations between UI elements and their implemented event handlers is, however, hidden by the OS. Without analyzing the source code, it is impossible to acquire such relationships without using a rooted device or a modified version of the OS.

3 THE UI INTERACTION MODEL

Unlike model-based approaches (e.g., [3, 33]) which extract a specific model for each app under test, we propose a *crowd-based approach* which extracts a *universal model* for Android apps. We use *machine learning* to automatically learn a model of common UI behavior, as inferred 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.

We aim to identify different types of UI elements (*e.g., Button*, *TextField*, *Layout*, etc.) and the types of events (e.g., *Click*, *Long-Click*, *Scroll*, etc.) they respond to (if any). By using a crowd of apps, our goal is to extract sufficient information to represent how developers implement interactions with users. Our approach to generate the *UI Interaction Model* takes 4 steps, as illustrated in Figure 2

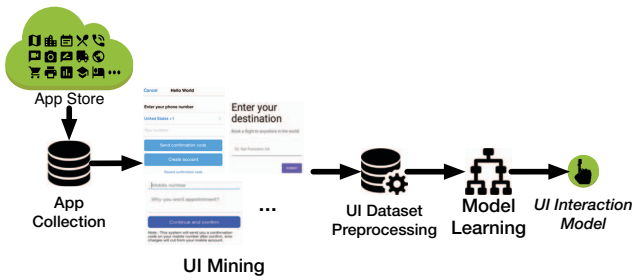


Figure 2: Model generation overview

3.1 App Collection

We start by collecting apps available in app stores. In particular, we downloaded 200 apps from the Google Play Store[2]. We opted for apps in the Google Play Store due to their widespread usage among users. To ensure that our sample covers different types of apps and

[2]The official market for Android apps.

actions, we selected some apps among the top-10 in each of the 32 categories, excluding games, as in [20][3].

To be statistically representative of app behavior throughout the whole app store, the apps used to mine the static model should have been randomly selected. However, users expect to interact with different apps in a similar way and thus apps tend to follow standard design practices [1]. We opted to use apps which are available in most devices – Facebook, YouTube, Instagram and Snapchat are, for example, available on more than 50% of the mobile devices [15] – as well as rely on the usage of common practices and Android design guidelines [7, 8] to generate a model which is representative of industry-standard behavior.

3.2 UI Mining

We then statically analyze each individual app to extract their UI elements (or Widgets in Android terms) and their associated code. Since we selected commercial apps, we cannot access the source code to map UI elements and event handlers. Therefore, we use the static analysis tool Backstage [9] to analyze the apps' binaries (*APKs*).

For each UI element we extracted the following features: *UI type*, *parent element*, *children elements*, *text*, and *event handler*. Note that some of the features may not have a value. Also, static analysis cannot guarantee the mapping between all UI elements and event handlers. For example, some widgets are created with dynamically loaded content.

The resulting dataset contains 119,397 unique UI elements. Approximately, 9% of the elements have an event attached, *i.e.,* react when the user interacts with them. This constitutes a challenge for dynamic exploration testing tools (such as Monkey [6], Droidmate [19], Droidbot [22], etc.) which randomly interact with elements in the screen. These tools perform many useless actions, that is, they select elements that do nothing. This means that computation time is wasted and efficacy is decreased because many widgets remain unexplored.

3.3 UI Dataset Preprocessing

Before learning a model from the mined UI data, we need to preprocess the raw UI information extracted by Backstage. We perform two tasks: *event propagation* and *UI type refinement*.

Event propagation. For each UI element in the dataset which lacks an attached event, we assign the event of its parent widget if it exists. This propagation is similar to workings of the Android OS event handling mechanism. When a UI element receives an event, the OS passes the event to the children 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.

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.

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

We map widget types with the standard UI 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, are considered as native. The custom widget types are mapped to native ones by computing string similarity distance (*i.e.,* Levenshtein Distance) [32]. We do not evaluate if the custom components are subclasses of the native Android ones because some companies opt to inherit from the root *View* class, instead of from a specialized Android version, however, even in these situations the name of components is 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 3a shows the distribution of the top-10 most frequent widgets in our dataset. We can observe that *LinearLayout* and *TextView* are the most used widgets across apps.

From our test dataset, Backstage identified 6 different types of event listeners, whose distribution is shown in Figure 3b. 93% of widgets which have an event implement the *onClick* event listener.

3.4 Model Learning

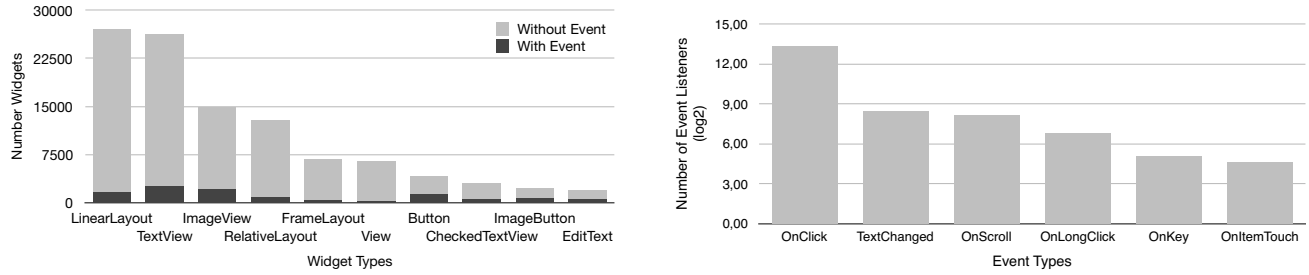
To identify the probabilities of UI elements correlating with events, we use a *Random Forest* [11] machine learning algorithm[4]. *Random Forest* consists of multiple independent tree-based learners which classify an instance by bagging the results of the individual classifiers. From our UI dataset, we use the existence of an *event* as class label to predict, and the remaining features as independent variables to learn. We discarded the event type in our current model because the tools used for evaluation support only *clicks* and *long clicks* and widgets with long click represented 1% of the identified events.

Among the different algorithms available, we chose *Random Forest* because it is fast, provides good accuracy results in general, and it is able to 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 [21]. *Random Forest* provided the best results.

Our dataset is inherently unbalanced, i.e., the majority of widgets lack events. This is due to incompleteness of the static analysis and/or the apps create a large number of widgets dynamically. The dataset contains a 1:9 ratio of widgets with and without event. Figure 3a illustrates the distribution of type of widgets with and without event. For example, 93.65% of *LinearLayouts* lack events attached. On the contrary, the 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 statically resolved without over-approximation. This is the rational 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 true and false class labels (1:1 ratio). We select all

[4]We use the Random Forest implementation provided by Weka: https://www.cs.waikato.ac.nz/ml/weka/



(a) ***Distribution of widgets with events*** **(b)** ***Listeners***

Figure 3: Distribution of widgets and event listeners in our dataset

10,938 widgets with an associated event, and a random subset of 10,938 widgets without an associated event.

4 GUIDED DYNAMIC EXPLORATION

To automate app testing, different input generation techniques and tools are available in the literature [23]. These techniques share the common goal of exploring the behavior of apps to discover potential faults. Nevertheless, they differ in the strategies to generate inputs and explore apps.

Automated testing techniques basically run a subject app and generate UI events to explore the app. The exploration strategies can be *random* or *systematic* (*i.e.,* guided by a model of the app). The challenge is to generate as many relevant inputs as possible to maximize the coverage of the exploration.

In this section, we illustrate how the *UI Interaction Model* inferred statically can be used dynamically to effectively direct the exploration of testing tools.

4.1 Consuming the *UI Interaction Model*

The *UI Interaction Model* predicts the likelihood of a UI element to possess an event listener. We use this probability to bias the input generation towards those UI elements which have higher chances to react to UI interactions.

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

To overcome this challenge, we use the *UI Interaction Model* to calculate the probability of each UI element being classified as True (has event) and False (no event). We assign as the fitness of the UI element the probability of having an event. Table 1 shows an example of fitness values assigned by our model to the elements of the *Change Log* screen.

Table 1: 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							

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%.

4.2 Fitness-Proportionate Selection Strategy

To select UI elements taking into consideration their fitness values and thus to bias explorations in favor of relevant UI elements, we use a *fitness proportionate selection* strategy from genetic algorithms.

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, $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, 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%: $p_{ButtonOK} = \frac{0.900}{2.250} = 40\%$.
This method simulates a roulette wheel with sectors of size proportional to p_i. Selecting an element is equivalent to choosing randomly a point on the wheel and locate its sector.

We execute the traditional version of the fitness selection algorithm n times and then *pick the most frequently selected individual*. This allows us to configure how likely widgets with low-fitness are expected to be selected. By selecting lower values of n, we increase the probability of choosing a low-fitness widget, as well as, by selecting a larger n, the widgets with higher fitness are more likely to be selected[5]. We experimentally defined $n = 10$.

The pie charts in Figure 1 (right) illustrates 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-biased random* approaches. In the biased approach, the *OK* button possess 40% selection probability, despite representing only the 5% of the screen elements.

4.3 Fitness Boost

The prediction power of the model 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 by 100%.

[5]The selection probability of the fitness proportionate selection are not kept for $n > 1$, instead the probability is calculated by a multinomial distribution distribution as demonstrated in [17].

The idea behind this boost is to give unexplored widgets the chance of being explored at least once. Before performing each action, the fitness value of each unexplored element is recalculated to 100%. Following this approach, once the "potential" elements (according to the model) are explored, the weaker elements will have higher probability of being selected at least once.

Considering the example presented in Figure 1, after the *OK button* has been clicked, its probability decreases to 25% while probabilities of clicking a label or layout increases to 41% and 33%. Each time a new element is selected, the probability of selecting the *Ok button* increases, until it again reaches 40% when all widgets in the screen have been explored one single time.

5 EMPIRICAL EVALUATION

In this section, we present empirical experiments to evaluate the applicability and effectiveness of our crowd-based approach to test Android apps. In particular, we aim to answer the following research questions:

- **RQ1.** Is the *UI Interaction Model* representative of normal app behavior? (Section 5.2)
- **RQ2.** Does the *UI Interaction Model* (learned statically) effectively predict events dynamically? (Section 5.3)
- **RQ3.** Is the *crowd-based dynamic exploration* more effective than the random and model-based explorations? (Section 5.4)

5.1 Experiment Set Up

5.1.1 Testing Tools. To perform the experiments, we apply the inferred *UI Interaction* model (cf. Section 3) into two existing Android testing tools: Droidmate [20] and Droidbot [22]. Droidmate 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 as Droidmate-M and Droidbot-M to the tools extended with our model.

5.1.2 Devices. To run the experiments, we use 4 mobile devices: Nexus 5X, Nexus 9, Nexus 6 and Pixel C, all running Android 7.1 (API 25). All tests of the same app are performed in the same device model to prevent inconsistencies coming from different device behaviors.

5.1.3 Benchmark Apps. We downloaded 17 new apps and installed them in the devices. To have a varied and representative sample, we select apps from *Google Play Store* and *F-droid*[6]. 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, while the use of commercial apps aims to illustrate that our approach improves tests even when no source code is available. The list of open source apps was extracted from [14], with some extra apps taken directly from F-Droid. Both open source and commercial apps were selected according to the limitations of the test generators used in the experiments.

[6]F-Droid is an open source repository of Android apps. https://f-droid.org

Table 2 summarizes the set of benchmark apps. For each app we provide the name, source, number of statements, number of widgets and number of events. To extract the number of statements we instrumented each app. The widgets and events were statically extracted with the Backstage tool.

In the remainder of this section, we present the findings for the three research questions. For each question, we describe the motivation, approach and results.

5.2 RQ1. Model representativity and generalization

5.2.1 Motivation. 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.

5.2.2 Approach. For this purpose, we assess the model using *10-fold cross-validation* on the training dataset (cf. Section 3) 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, thus, this self-test allows us to verify whether and how much the model is representative of app behavior.

5.2.3 Results. Figure 4 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 actually do have event handlers) is 68%, whereas the overall *recall* (the percentage of classes with event handlers that are correctly predicted as such) is 75%.

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

We also evaluate the original dataset before performing subsampling (cf. Section 3.4). In this case, the model presents a precision and recall superior to 90%. However, this is due to the 9:1 ratio of widgets without and with event in the dataset. The classifier simply classified all widgets as *without event*. Balancing the dataset avoids biased results.

Based on the findings of this experiment we conclude that the *UI Interaction Model* possess enough features to correctly predict events on the training set.

[7]These apps crashed when being evaluated with Backstage.

Table 2: Set of benchmark apps

App	Source	Downloads	Category	#Stmts	Widgets	Events
Alogblog[7]	F-Droid	-	Internet	428	-	-
KeePassDroid (2.0.6.4)	F-Droid	-	Security	43	169	0
Munch (0.44)	F-Droid	-	Internet	8084	387	0
BART Runner (2.2.6)	F-Droid	-	Navigation	8125	170	5
Jamendo (1.0.4)[7]	F-Droid	-	Music	9347	-	-
2048 (2.06)	F-Droid	-	Games	168	3	1
DroidWeight (1.3.3)	F-Droid	-	Sports & Health	4279	63	22
Pizza Cost (1.05-9)[7]	F-Droid	-	Money	1240	-	-
Mirrored (0.2.9)	F-Droid	-	Internet	2475	29	0
Easy xkcd (5.3.9)	F-Droid	-	Internet	13768	265	6
Dialer2 (2.90)	F-Droid	-	Phone & SMS	2005	55	19
PasswordMaker (1.1.11)	F-Droid	-	Security	4378	177	30
Tomdroid (0.4.1)	F-Droid	-	Writing	2727	21	0
World Weather (1.2.4)	Play Store	1k-5k	Weather	4116	205	0
SyncMyPix (0.16)	Play Store	250k-500k	Social	10084	81	15
Der Die Das (16.04.2016)	Play Store	500k-1M	Learning	3225	69	0
wikiHow (2.7.3)	Play Store	1M-5M	Books and Reference	3703	183	7

The features *widget, parent and children types*, extracted statically from a multitude of apps, can be used to correctly predict if unseen widgets have event handlers associated.

5.3 RQ2. Prediction Power of the UI Interaction Model

5.3.1 Motivation. The *UI Interaction Model* is obtained by performing static analysis in 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. In addition, the inferred knowledge can be incomplete. We aim to investigate if the static knowledge represents dynamic behavior.

5.3.2 Approach. For this purpose, we evaluate the performance of the model at runtime. 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 explorations. We consider *effective actions* the ones 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, and each execution trace contains hundreds of actions; it is infeasible to manually assess each individual action. For this reason, we use an heuristic to compute automatically the number of *effective actions*. Intuitively, if after an action there is a reaction in the app, then a change will happen on the app screen.

We consider an action as *effective* if the screenshots before and after the action are different. To measure similarity between screenshots, we apply an image processing approach similar to [28]. We perform an image subtraction for each pair of consecutive screenshots. If both screenshots in a pair are identical[8], 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 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 [16]. We perform a reduced manual evaluation with the subject apps, and we conclude that screenshot similarity is a suitable approach to identify *effective* and *ineffective* actions.

Droidmate provides the feature to capture screenshots after each action. Therefore, we use Droidmate to perform this experiment. We run Droidmate and Droidmate-M to test the 16 benchmark apps (cf. Section 5.1). 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 10 times and compute the average percentage of ineffective actions.

5.3.3 Results. In 13 out of 17 apps, the guided-exploration *significantly reduces the number of ineffective actions.* As shown in Fig.5, the number of ineffective actions varies among apps, since it depends on the app design.

On average, Droidmate 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.

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 basically all elements displayed on the screen respond to interactions. Thus, any random interaction has

[8]Note that we first remove the top part of the images corresponding with the device *'status bar'*. The status bar contains widgets such as clock and battery level which could evolve along the execution, thus producing false positives.

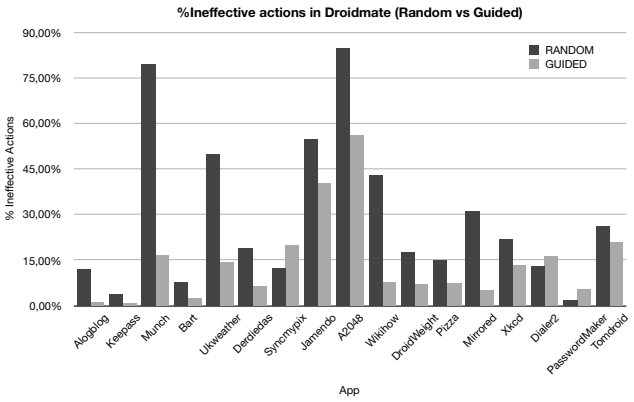


Figure 5: Comparison of *ineffective actions* in Droidmate between Random and Guided exploration

an effect. Since, the guided-exploration gives a boost to unexplored elements (cf. Section 4.3), it executed more ineffective actions, because all elements are triggered at least once, and those elements are less triggered by the random exploration.

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

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

5.4 RQ3. Effectiveness of the Approach

5.4.1 Motivation. The third question aims to quantify the effectiveness of the crowd-based approach in comparison with state of the art testing approaches.

5.4.2 Approach. We compare the performance of the tools Droidmate and Droidbot against the extended versions which use our model—i.e. Droidmate-M and Droidbot-M. Furthermore, we compare with Monkey tool [6]. Monkey (provided by Android) is the most popular tool to test Android apps. Monkey generates random UI events (*e..g,* clicks, touches) to stress apps, without taking into consideration the UI of the apps. Although Monkey implements the most basic strategy, previous research have shown that Monkey outperforms other sophisticated techniques [14].

Code coverage has been extensively used to determine the effectiveness of testing tools. Previous research has demonstrated that code coverage is a good predictor for fault detection [18]. Therefore, we select code coverage as metric to evaluate and compare testing strategies. Similar to [14], we compute statement coverage. To measure statement coverage, we use the *Androcov* tool[9] which is able to measure coverage without having access to the source code. The current implementation of *Androcov* only supports method coverage. We extended it to measure statement coverage by inserting a unique log instruction before each statement in the app bytecode instead of a single instruction in the start of each app method. This implementation will benefit future Android testing studies, which could measure coverage going beyond open-source apps.

We run the 5 tools with the 16 benchmark apps (cf. Table 2). Previous experimentations with Android apps have illustrated that testing tools reach maximum coverage peak after exploring between 5 and 10 minutes [14], [19]. Taking this data as starting point, we first run a exploratory phase with our subject apps and testing tools to determine the number of actions that saturate explorations (*i.e.,* point where the apps reach the maximum coverage and stay constant along time).

We configure Droidmate and Droidmate-M to execute 500 actions with each app. Each exploration takes ~25 minutes to complete. With the Droidbot tool, we could not complete explorations with more than 200 actions because the execution got stuck. To enable comparison, we configure Droidbot and Droidbot-M to execute 100 actions with each app. Finally, Monkey is much faster than Droidmate and Droidbot because it lacks UI analysis (*i.e.,* simply generates random inputs events to the device). Therefore, we select 5,000 actions which correspond with approximately 5 minutes of exploration (as saturation point stated by previous work).

For each app and tool, we run the exploration 10 times and report the median coverage obtained across all apps.

5.4.3 Results. Guided exploration is more effective than the original random and model-based explorations in Droidmate and Droidbot. Figure 6 reports the median coverage that each tool achieved with the benchmark apps.

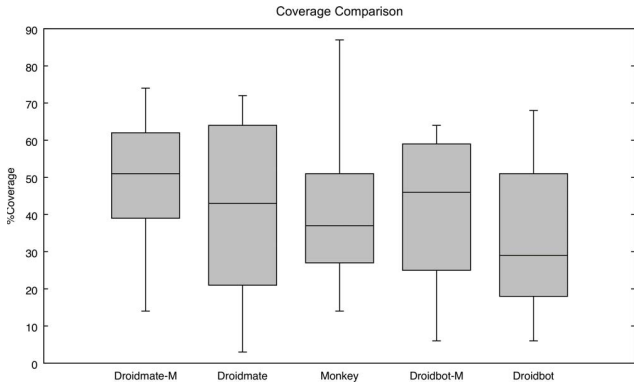


Figure 6: Coverage comparison of *guided* and *random* explorations over the benchmark apps

Note that with the Droidbot tool, the apps *PasswordMaker* and *Tomdroid* (cf. Table 2) crashed during execution and Droidmate stopped. Thus, the results reported in Figure 6 for Droidbot and Droidbot-M exclude those two apps.

While the median coverage of Droidmate 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.

[9] https://github.com/ylimit/androcov

Droidbot obtains a coverage percentage increase of 55.30% when using the guided strategy. From a median coverage of 29.87% goes up to 46.39%. In Droidbot the gain is bigger (than in Droidmate) because the actions space is bigger. When a widget is found, Droidbot choses between 5 different actions (*i.e.,* click, touch, swipe up, swipe down, swipe left, and swipe right). Using the guided-exploration the search-space is pruned with the widgets most likely to have an event. We are currently working on experiments to incorporate the event type in the model. This experiment demonstrates that the guided exploration improves the performance of random explorations.

In addition, Monkey obtained a median coverage of 37.2%. First, we can observe that both Droidmate and Droidmate-M outperform the median coverage of Monkey. Second, Monkey performs better than Droidbot. Nevertheless, when using the guided strategy, Droidbot-M beats Monkey and also Droidmate.

*The state of the art tools Droidmate and Droidbot experience a **coverage percentage increase of 19.41% and 43.03%** on average when using the guided strategy across all apps.*

If we go into the details of particular apps, we observe that there are 4 apps (*Aalogblog*, *KeePassDroid*, *2048* and *Tomdroid*) where the random and guided explorations obtain the same coverage. Since the 2 different testing methods achieve the same statement coverage, it means it is the maximum that can be achieved. As illustration, Figure 7a shows the coverage evolution of the exploration in Droidmate with random and guided exploration in the *2048* app. We can observe that even if both the guided and random exploration converge into the maximum coverage, the guided exploration covers faster. We manually check if with manual exploration it is possible to achieve higher coverage. In fact, there are login screens which the testing tools fail to pass, as well as swipe actions which the testing tools failed to execute.

We further explore three cases. First, we show a case where the random exploration outperforms the guided-exploration, and two cases where the guided-explorations win. Figure 7b shows the coverage evolution in the *DroidWeight* app, where the random exploration outperforms the guided one. We can observe that the random exploration was consistently better throughout the whole exploration. However, while the random coverage's curve has flattened for approximately 10 minutes, the guided exploration was able to explore new widgets. Finally, both strategies converge.

In the *Jamendo* app (Figure 7c), 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. In addition, the *PasswordMaker* app shown in Figure 7d illustrates a situation where the guided exploration outperforms the random one from the initial screen.

As we have previously discussed, the performance of the approach highly depends on the design of the apps.

Figure 8 reports the coverage evolution along the exploration of Droidmate and Droidmate-M. The graph shows the average coverage across all benchmark apps over 10 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% in comparison with the random exploration. This becomes specially relevant in mobile ecosystems, where time is a critical success factor. App updates are frequent [27], by reducing testing time, release cycles can be accelerated.

The guided-exploration speeds-up testing.

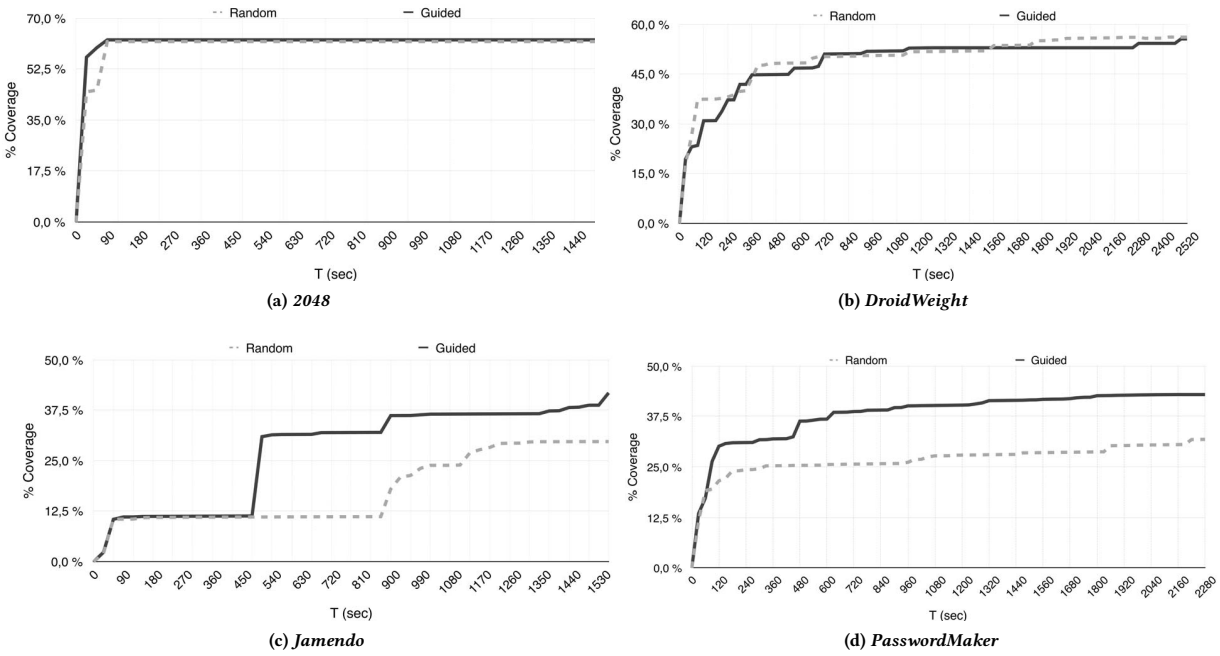
In addition, our experiments have demonstrated the *applicability* of our model. We have applied the model into two different testing tools (Droidmate and Droidbot) which are implemented with different programming languages (*i.e.,* Kotlin and Python), and follow different testing strategies. The extensions 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 first has to mine a universal model, as described in this paper. Even though this model can be reused again and again (unless the essential behavior of Android widgets changes over time), one may ask whether this effort is actually necessary. In particular, one may ask whether it would not be easier 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 Droidmate (*i.e.,* random exploration) performed better than Droidmate-M, which indicates that this app was not accurately represented by the crowd-based model. We statically extracted the widgets and events *from the* DroidWeight *app only* and trained a random forest classifier, similarly 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 Sections 5.3 and 5.4; similar to the previous experiments we execute 500 actions and 10 runs.

When using a single-app model, the amount of ineffective actions drops from 7.05% to 3.23%, i.e., 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%. This difference is explained by the limitations of the static analysis. 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*



(a) *2048* (b) *DroidWeight*

(c) *Jamendo* (d) *PasswordMaker*

Figure 7: Average Coverage Evolution in Droidmate with *guided* and *random* explorations

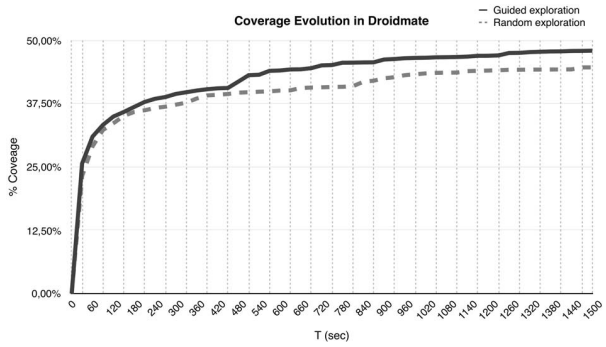


Figure 8: Average coverage evolution in Droidmate with *guided* and *random* explorations over the benchmark apps

limitations of static analysis and, thus, ***achieve less code coverage****.*

6 DISCUSSION AND THREATS

The presented approach and experimental evaluation present several limitations and threats to validity.

Regarding external validity, our experiments have demonstrated evidence that the guided exploration using crowd knowledge significantly improves testing performance of state of the art tools (Droidmate, 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, containing commercial and open-source apps, having different sizes and categories. In addition, we applied the approach to two testing tools which implement different exploration strategies and use different programming languages.

The proposed approach is implemented on the Android platform because currently it is the most widespread mobile OS (having +88% of market share). Nevertheless, the conceptual foundations presented in this paper 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 5), but also due to limitations of the tool. 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. A more precise measurement can be obtained by using source code instead of bytecode for

instrumentation and by measuring coverage on Javascript, dynamic and native code.

7 RELATED WORK

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

There is a prolific research on the area of *Automated Test Input Generation* in mobile apps. Input generation tools are commonly classified into three categories: *random testing*, *model-based testing*, and *systematic testing*.

Random testing tools create random chains of events to explore apps' behavior. These approaches are unlikely to complete tasks that demand human intelligence, such as properly filling a registration form; however they are effective to test apps' robustness and error handling capabilities. This strategy is employed by Monkey [6], the test generation shipped with the Android development toolkit, to generate a predefined number of random events. *Dynodroid* [24] and Droidmate [19] use a biased random approach. These tools randomly generate events to interact with widgets which have been least explored. Our approach differs from the traditional random exploration by giving preference to UI elements which have higher probability to have event handlers attached. However, in a scenario where all UI elements have the same probability to contain an event, the approach behaves equal to a purely random.

Model-based testing tools infer models from applications using static and/or dynamic analysis and use them to generate test cases. *GUIRipper* [2] and *MOBIGuittar* [3] dynamically traverses an app's GUI and create a state machine model which is later used to generate test inputs. *ORBIT* [31] follows an approach similar to *GUIRipper* and *MOBIGuittar*, but uses static analysis to reduce the number of GUI elements to test. *SmartDroid* [33] also exploits static analysis to generate activities and function call graphs to identify paths which should be explored. *SwiftHand* [13] applies machine learning to create a model of the app which is used to generate inputs with the aim of visiting unexplored app states. A^3E [10], *CuriousDroid* [12], and *Droidbot* [22] dynamically generate finite state models which capture transitions among activities to guide the exploration. Our approach differs from these approaches by generating a universal model from a multitude of apps which can be reused among apps and testing tools. With this difference we are able to overcome two common setbacks of model-based approaches. First, we are able to interact with apps which dynamically load content—and, thus, cannot be fully 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 exhaustively test applications, or to generate tests which trigger specific behaviors. *EvoDroid* [25] and *Sapienz* [26] use search-based algorithms—evolutionary or combined with random fuzzing—to improve test coverage. *ACTEve* [4] and *IntelliDroid* [30] apply symbolic execution to generate feasible event sequences for Android apps in order to trigger specific behaviors. Tailoring specific behaviors mostly demand static analysis to identify the path of the app's execution flow which must be followed. Testing completely a real app 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.

8 CONCLUSION AND FUTURE WORK

This paper presents an approach to guide app test generators towards UI elements that are most likely to be reactive. We 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 which have higher probabilities of success.

We applied the approach into 2 existing state of the art testing tools: Droidmate and Droidbot. 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 on code coverage 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, we did not implement the fine-grained identification of the specific kind of events. We are currently working on incorporating the prediction of the types of events into the model. This extension will potentially improve even more the performance of the approach.

Hybrid and adaptive models. In Section 5.4 we illustrated the benefits and limitations of using a single-app model. As future work, we plan to explore *hybrid models* which combine crowd knowledge with specific-app knowledge as well as with models which can be fine tuned (adapt) during dynamic analysis using, for example, reinforced learning.

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 underlaying app code, a user is capable of recognizing 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, thus leading to bad user experiences.

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

Droidmate-M, Androcov-Statement and Dataset:
https://github.com/uds-se
Droidbot-M: https://github.com/natanieljr/droidbot

Acknowledgments: This work was funded partially by and European Research Council grant (G514111401) and partially by a Deutsche Forschungsgemeinschaft grant (D514111409).

REFERENCES

[1] Khalid Alharbi and Tom Yeh. 2015. 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*. ACM, 515–524.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *ASE 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 258–261.

[3] D Amalfitano, A R Fasolino, P Tramontana, B D Ta, and A M Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*.

[5] Android. 2017. UI Overview. (2017). https://developer.android.com/guide/topics/ui/overview.html

[6] Android. 2017. UI/Application Exerciser Monkey. (2017). https://developer.android.com/studio/test/monkey.html

[7] Android. 2018. Android Design Guidelines. (2018). https://developer.android.com/design/index.html

[8] Android. 2018. Material Design for Android. (2018). https://developer.android.com/design/material/index.html

[9] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. 2017. Detecting behavior anomalies in graphical user interfaces. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 201–203.

[10] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA '13 Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641—-660.

[11] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[12] Patrick Carter, Collin Mulliner, Martina Lindorfer, and William Robertson. 2016. CuriousDroid : Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC'16)*.

[13] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *OOPSLA '13 Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 1–30.

[14] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.

[15] comScore Inc. [n. d.]. The 2017 U.S. Mobile App Report. ([n. d.]). https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/The-2017-US-Mobile-App-Report

[16] Alan Dix. 2009. Human-computer interaction. In *Encyclopedia of database systems*. Springer, 1327–1331.

[17] PR Freeman. 1979. Algorithm as 145: exact distribution of the largest multinomial frequency. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 3 (1979), 333–336.

[18] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 72–82.

[19] Konrad Jamrozik and Philipp Von Styp-rekowsky Andreas. 2016. Mining Sandboxes. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on* (2016). https://doi.org/10.1145/2884781.2884782

[20] Konrad Jamrozik and Andreas Zeller. 2016. Droidmate: a robust and extensible test generator for android. In *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, 293–294.

[21] Sotiris B Kotsiantis, Ioannis D Zaharakis, and Panayiotis E Pintelas. 2006. Machine learning: a review of classification and combining techniques. *Artificial Intelligence Review* 26, 3 (2006), 159–190.

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot : A Lightweight UI-Guided Test Input Generator for Android. *2017 IEEE/ACM 39th IEEE International Conference on Software Engineering* (2017). https://doi.org/10.1109/ICSE-C.2017.8

[23] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 399–410.

[24] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. 224.

[25] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. 599–609. https://doi.org/10.1145/2635868.2635896

[26] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[27] Stuart McIlroy, Nasir Ali, and Ahmed E. Hassan. 2016. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 1346–1370. https://doi.org/10.1007/s10664-015-9388-2

[28] Julia Rubin, Michael I Gordon, Nguyen Nguyen, and Martin Rinard. 2015. Covert Communication in Mobile Applications. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. 647–657.

[29] Statista. [n. d.]. Google Play: number of available apps 2009-2017. ([n. d.]). https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[30] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 21–24. https://doi.org/10.14722/ndss.2016.23118

[31] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13*. 250–265.

[32] Li Yujian and Liu Bo. 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007), 1091–1095.

[33] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, and Xiaorui Gong. 2012. 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)*. 93–104. https://doi.org/10.1145/2381934.2381950