Learning User Interface Element Interactions

Christian Degott
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
christian.degott@cispa.saarland

Nataniel P. Borges Jr.
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
nataniel.borges@cispa.saarland

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.saarland

ABSTRACT

When generating tests for graphical user interfaces, one central problem is to identify how individual UI elements can be interacted with—clicking, long- or right-clicking, swiping, dragging, typing, or more. We present an approach based on *reinforcement learning* that automatically learns which interactions can be used for which elements, and uses this information to guide test generation. We model the problem as an instance of the *multi-armed bandit problem* (*MAB problem*) from probability theory, and show how its traditional solutions work on test generation, with and without relying on previous knowledge.

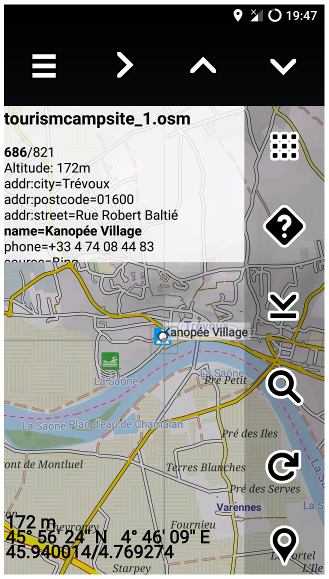
CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Dynamic analysis*; • **Human-centered computing** → Graphical user interfaces; Smartphones.

KEYWORDS

Test generation, User interactions, Multi-Armed Bandit problem, Android

ACM Reference Format:
Christian Degott, Nataniel P. Borges Jr., and Andreas Zeller. 2019. Learning User Interface Element Interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330569



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

Figure 1: 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.

1 INTRODUCTION

To explore the functionality of a mobile app (including its errors), *automated test generators* systematically identify and interact with its user interface elements. One key challenge is to synthesize inputs which effectively and efficiently cover app behavior. This is a non-trivial problem, as a test generator not only has to infer the set of user interface elements, it also has to infer which interactions are possible with each element.

As an example, consider Figure 1, 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 open 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.

Recent research attempts to emulate such knowledge for test generators by gathering knowledge from other apps and transferring this knowledge to new apps. *Static techniques* [10] mine associations between UI elements and their interactions from the most common applications, and can learn, for instance, that "button" UI Elements typically accept "click" interactions. *Dynamic techniques* [21] learn these actions from dynamic executions, and again apply a pre-approximated probability distribution mined from several apps.

Both these approaches, however, are strongly *biased* towards the distribution originally 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 1—a "layout" object:

- 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 was 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 pre-conceived notion of *likely* interactions, while *gradually including less likely actions* if the likely actions do not succeed.

To this end, we approach test generation as an instance of the *multi-(or N-) armed bandit problem* [20] (*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.

The remainder of the paper is organized as follows. After discussing the background (Section 2), we make the following contributions:

(1) We introduce test generation as an instance of the *MAB problem* (Section 3), formulating two strategies for reinforcement learning without prior knowledge. To the best of our knowledge, this is the first time the *MAB problem* is used as a model for test generation.
(2) We show how to enhance our reinforcement learning models with *statically trained models* from previous research to show how to integrate pre-approximated probability distributions (Section 4). To the best of our knowledge, this is the first time a reinforcement learning approach is not only used in test generation to train a model, but also to gradually adapt the model to the application under test.
(3) We *evaluate* both strategies (Section 6), and show that
    - **Reinforcement learning can be used to more effectively test apps.** Compared to a statically gathered crowd-model, the average coverage increases by more than 18%.
    - **Reinforcement learning without a priori knowledge outperformed those with a priori knowledge.** The difference in coverage is up to 8%. This shows the advantage of the *MAB problem* approach over pre-mined models.
    - **Adding reinforcement learning to a statically mined model improved coverage.** In our experiments, the addition of reinforcement learning to a statically mined model lead to 20% coverage improvement.

After discussing limitations and threats to validity, we close with conclusion and future work.

2 BACKGROUND

The graphical user interface (GUI) of an Android app is hierarchically constituted of graphical and structuring elements, commonly referred to as widgets. Android apps are event driven,all interaction between the app and the world happens through events. To test an Android app is to generate a sequence of events, such as touching or swiping on different locations. It is not possible to, without a modified OS version or support from static analysis, be certain that an event will trigger any action in the app. This creates a major challenge for most testing strategies, as many visible UI elements are passive, i.e., used only for layout and appearance.

Borges *et al.* [10] gathered static models from a set of apps and reused this knowledge on new apps dynamically. The authors' goal was to emulate how humans can transfer knowledge they have of previous apps, while testing new ones. Their static model predicted how likely was for each UI element on a UI to trigger an event. While their experiments showed up to 43% improvement over state-of-the-art test generators, it still yields a major limitation: not all behaviors can be captured statically. In addition, the effectiveness of their approach was tightly associated to how similar the behavior of the app under test is to their test set.

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

We approach test generation as an instance of the *MAB problem* [24]. 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 one. 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 [8]. 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* [9].

The *MAB problem* has been addressed with *reinforcement learning* techniques, such as *ϵ-greedy* [31] and *Thompson sampling* [28], with good results [13, 29].

The ϵ-Greedy strategy, illustrated in Figure 2, considers a predefined 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 [26]—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 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 3a.

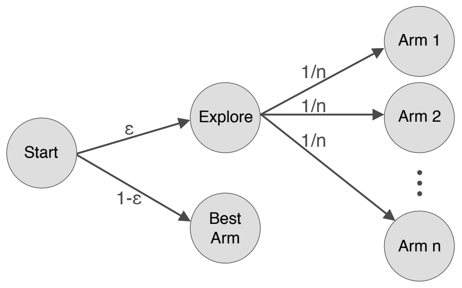


Figure 2: Overview of the ϵ-Greedy approach.

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 3b and Figure 3c. 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*).

3 TESTING WITH REINFORCEMENT LEARNING

To test an app, a test generator is presented with UI states containing multiple elements (widgets). 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 own probability of improving testing. Our goal is to dynamically learn how to interact with an app, that is, not only to identify which UI elements should be triggered, but also which kind of interaction should be used, thus 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*.

3.1 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. We, thus, 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 state S the set of UI elements available on the app screen at a specific time. To pull an arm is equivalent to perform the action 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 individual 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 [10] 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.

Reward. 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 which 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 to not restrict our approach to any specific apps or environment. We therefore consider an action *effective* if the screens before and after performing the action are different.

Our heuristic relies on design principles [16] which dictate that 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 which do not trigger UI changes are frequently classified as possible misbehaviors [25].

3.2 ϵ-Greedy Strategy

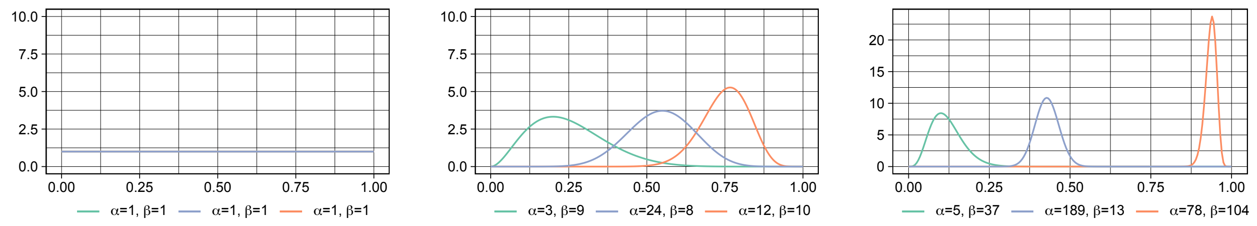
We modeled the ϵ-Greedy approach according to Algorithm 1. We first initialize the current (*wins*) and total (*trials*) counters and probabilities for all classes C and action types A (line 2-5). Since our algorithm starts with no previous app knowledge, we initialize all *wins* and *trials* with 0.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

3.3 Thompson Sampling

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 highest value (lines 6-9). We then perform the action a on the widget w, obtain a reward r (line

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



(a) Initial distribution (b) Distribution after a few updates (c) Distribution after some more updates

Figure 3: Three B-distributions changing over time

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*)

```
Input: $\epsilon \in (0,1)$
 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
```

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 encodes the accumulated knowledge.

4 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. In addition, reinforcement learning can be used alongside previously existing approaches, to enhance previously gathered models with information about the app under test.

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
```

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:         $r \leftarrow e.w$.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
```

4.1 Previous Knowledge With Reinforcement Learning

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.

4.2 Integrating Reinforcement Learning to Statically Gathered Models

The principles of reinforcement learning can be applied to other test generation strategies for a 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* [10]. We extend the authors original approach with our reinforcement learning model so that the test generator can adapt to cases in which the original model was not so effective.

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 the original stochastic select algorithm[2] from Borges et al. [10] replacing the *crowd-model* for our extended version (Line 10).

We then perform the action 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 of each action type and class allow the *crowd-based* model to be adjusted according to the app behavior.

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.

```
Input: $n > 0$
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
```

5 IMPLEMENTATION

We implemented our approach as plug-ins for DroidMate-2 [11], an open-source Android test input generator that can be used out of the box on Android devices running versions 6.0 to 8.0 without app source code, root privileges or OS modifications.

[2]The original stochastic select algorithm from Borges et al. is a sequence of roulette wheel selections in which the most selected element is chosen.

DroidMate-2 is able to monitor statement coverage during execution and it internally handles all device communication, UI modeling and extraction. DroidMate-2 interacts with the accessibility service to obtain the available UI elements as well as to act on them. Its public API allows us to directly interact with UI elements instead of sending events to coordinates.

Since Android does not provide a unique identifier for UI elements, DroidMate-2 uses a heuristic to uniquely identify them, based on the textual content of the UI element, if any, or on the UI element's image otherwise. It then uses these UI elements as an heuristic to uniquely identify a UI state. We use this metric to determine if an action was effective. If the state before the action is different than the state after, we consider that the action was effective.

We extended the original set of capabilities from DroidMate-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.

6 EVALUATION

We present a set of experiments to gather empirical evidence of the benefits of using reinforcement learning to test Android apps. In particular, we aim to answer the following research questions:

RQ1. Can reinforcement learning be used to more effectively test apps?

RQ2. Is knowledge learned from the app under test more beneficial to testing than static models from other apps?

RQ3. Can reinforcement learning be used to enhance static models?

In our experiments we used code coverage to measure the exploration quality as code coverage has been shown to be a good predictor for a test suite quality [18]. We obtained the coverage through DroidMate-2 native instrumentation mechanism. Moreover, we reused the set of benchmark apps from [10], shown in Table 1, 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*[3], the official market for Android apps, and from *F-droid*[4], an open-source repository of Android apps. Compared to the original works, we excluded the apps Alogblog, Jamendo, DroidWeight, Tomdroid and SyncMyPix as they either no longer work on newer versions of Android or were unavailable for download.

6.1 RQ1 – Reinforcement Learning

By guiding test generation towards more effective UI elements, previous research [10] showed that it was possible to achieve better tests. In this experiment we want to gather empirical evidence that it is unnecessary to collect data a priori, but that it is possible to learn 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

[3]https://play.google.com/store/apps
[4]https://f-droid.org
[5]This app crashed when being evaluated with Backstage.

Table 1: Set of evaluation benchmark apps

App	Source	Downloads	Category	#Stmts	Widgets	Events
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
2048 (2.06)	F-Droid	-	Games	168	3	1
Pizza Cost (1.05-9)[5]	F-Droid	-	Money	1240	N/A	N/A
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
World Weather (1.2.4)	Play Store	1k-5k	Weather	4116	205	0
Der Die Das (16.04.2016)	Play Store	500k-1M	Learning	3225	69	0
wikiHow (2.7.3)	Play Store	1M-5M	Books & Reference	3703	183	7

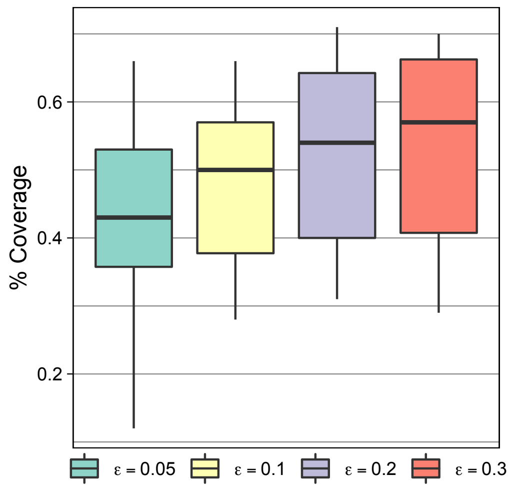


Figure 4: Results for the parameter tuning experiments for selecting ϵ

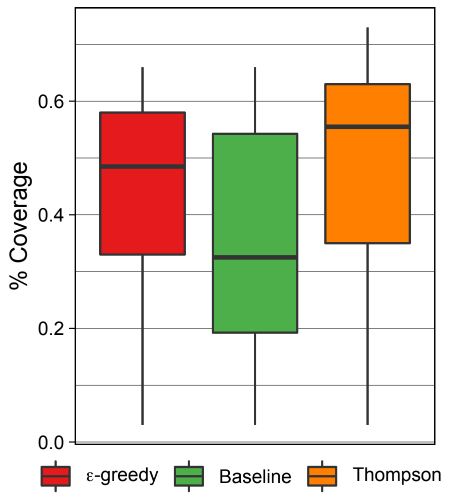


Figure 5: Comparison of statement coverage between Baseline, ϵ-Greedy, and Thompson

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 [10, 11] to outperform DroidBot [22], Monkey [3] and the original DroidMate on this same dataset.

We explored each of the 12 apps from our test dataset 10 times on Google Pixel 2 XL devices running Android 8.1 (API 27) and we obtained the average coverage from these tests. We opted for 10 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 to not significantly increase the coverage of random testing tools [11]. Moreover, we used the same amount of actions for both approaches since the reinforcement learning approaches do not have a meaningful performance penalty when compared against the baseline.

Parameter calibration. The ϵ-Greedy approach requires a value for ϵ to be determined. This value is used to determine the strategy exploration/exploitation rate and has a huge impact on its behavior. Before our experiment we performed a small-scale experiment to determine its value. We randomly selected 5 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 shown in Figure 4, we opted for $\epsilon = 0.3$.

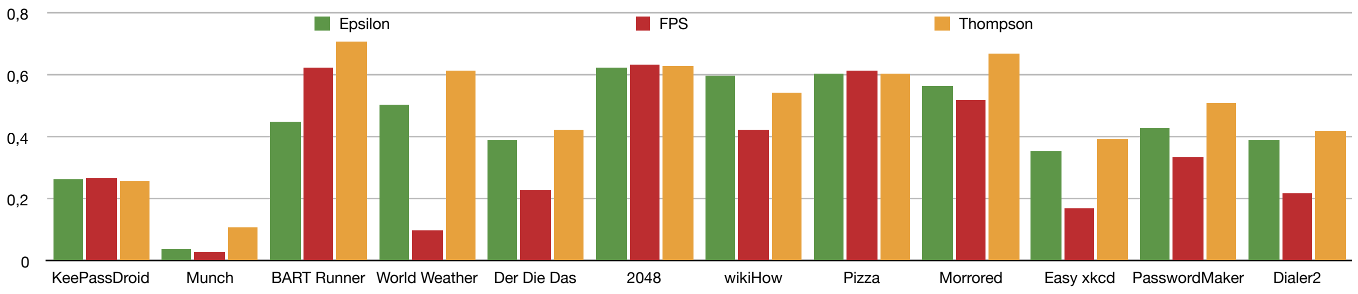


Figure 6: Per app comparison of statement coverage between Baseline, ϵ-Greedy, and Thompson

Results. The results of this experiment are shown in Figure 5. 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 6, Thompson obtains more coverage on 8 apps, ϵ-Greedy obtains more coverage on 1 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%.

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

6.2 RQ2 – Reinforcement Learning with A Priori Knowledge

Our previous experiment showed that reinforcement learning approaches can guide test generation towards more effective UI elements, leading to a 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 10 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 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 [10] as a priori knowledge, since it is representative of app behavior and, similar to our approach, can be used on any arbitrary app.

Parameter Calibration. To use a priori knowledge alongside our reinforcement learning approaches it is necessary to define a weight ψ for the model. Higher values of ψ reduce the reinforcement learning effect and increase the relevance of initial knowledge, smaller values give a higher significance to knowledge gained through reinforcement learning. To determine a value for ψ, we randomly selected 5 apps from the test dataset and explored them 4 times, using Thompson+K, for each of the following ψ values: 10, 20, 50, and 100. Based on coverage obtained by these tests, as shown in Figure 7, we opted for $\psi = 20$.

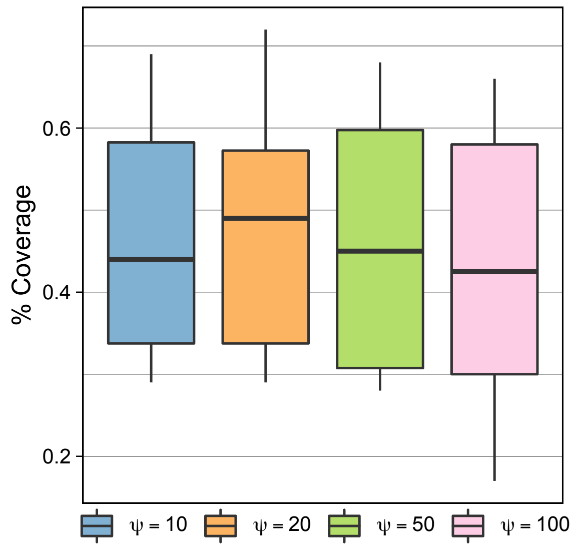


Figure 7: Results for the parameter tuning experiments for selecting ψ

Results. The results of our comparison between ϵ-Greedy, ϵ-Greedy+K, Thompson and Thompson+K are shown in Figure 9. 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, however, prevented bad random seeds from achieving specially bad results during testing, increasing the minimum overall coverage achieved. On a per app analysis, presented in Figure 8, ϵ-Greedy obtains more coverage on 2 apps, ϵ-Greedy+K obtains more coverage on 4 apps–including 3 draws against Thompson or Thompson+K. Thompson obtains more coverage on 8 apps–including 2 draws against ϵ-Greedy+K–and Thompson+K obtains more coverage on 1 app–drawing against ϵ-Greedy+K. To verify the statistical significance of our results we performed a Friedman's

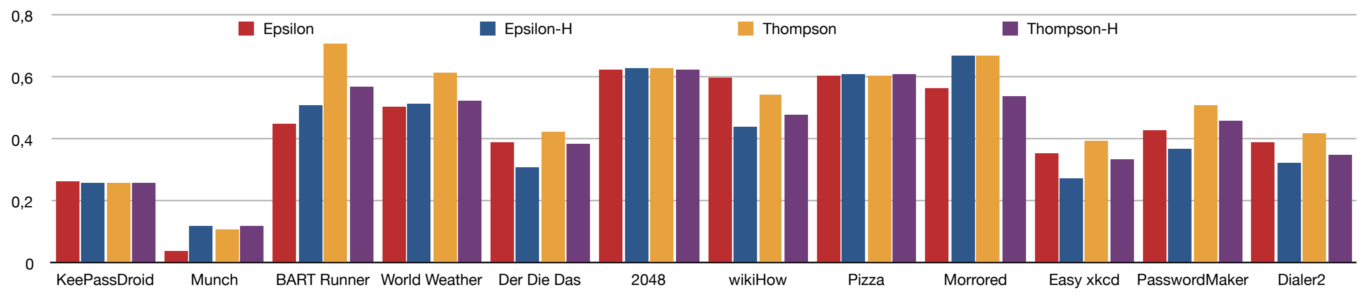


Figure 8: Per app comparison of statement coverage between BASELINE, ϵ-GREEDY, and THOMPSON

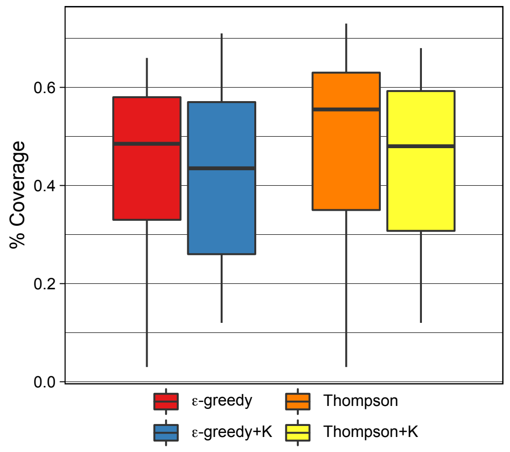


Figure 9: Comparison of coverage between ϵ-GREEDY, ϵ-GREEDY+K, THOMPSON and THOMPSON+K.

test. It resulted in a *p-value* of 0.00073, indicating that our results are significant at 5%.

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.

6.3 RQ3 – Extending Static Models With Reinforcement Learning

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 10 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 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.

Results. The results of our comparison between BASELINE and BASELINE+K are shown in Figure 10. 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, presented in Figure 11, BASELINE obtains more coverage on 2 apps and BASELINE+K obtains more coverage on the remaining 10 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%.

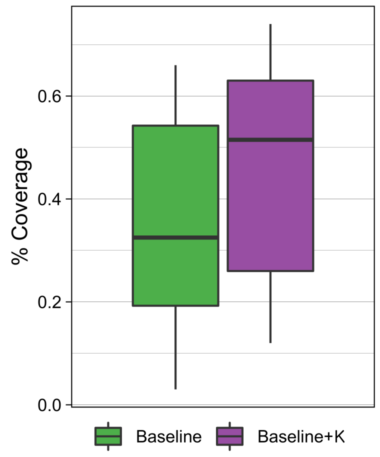


Figure 10: Comparison of coverage between BASELINE and BASELINE+K

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

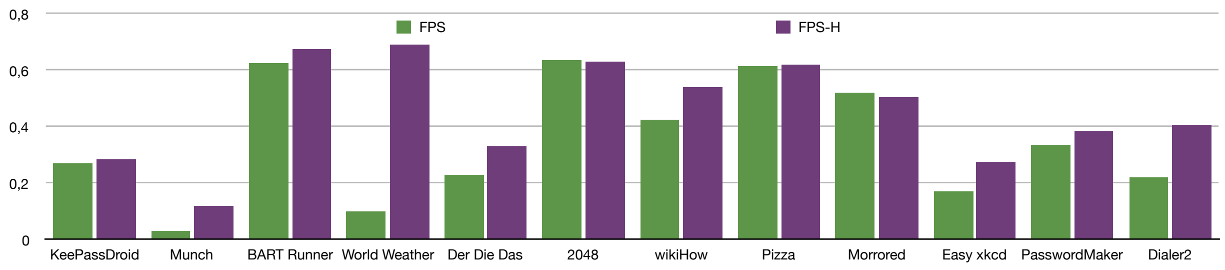


Figure 11: Per app comparison of statement coverage between Baseline, ϵ-Greedy, and Thompson

7 LIMITATIONS AND THREATS TO VALIDITY

Our approach and experimental evaluation has limitations and 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 (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 DroidMate-2 and, thus, inherits its limitations. Apps which cannot be run on DroidMate-2 cannot be run 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. It does not, however, analyze the semantics of a UI to input meaningful values on input fields or to perform tasks in a human-like way. Therefore, our approaches' maximum coverage is limited by the inherent limitations of random exploration strategies. Our ϵ-Greedy and Thompson approaches, however, could be applied alongside any systematic or model-based testing tool to assist them into prioritizing between two equally good options, which is now done at random. Finally, to exploit a priori knowledge we used the UI interaction model mined and trained by [10], thus, this data suffers from the same threats to construct validity as theirs: the extracted information for the UI interaction models is incomplete because of inherent limitations of static analysis and the actual tool, *Backstage* [4], 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 the ability to find faults and the code coverage of a test suite, the use of a curated repository of bugs could provide more accurate results regarding the effectiveness of the techniques.

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.

8 RELATED WORK

Automated test generation in mobile apps is an active research field. Test input generation strategies are commonly classified into three categories [15] *random*, *model-based*, and *explorative* strategies.

8.1 Random Strategies

Random strategies generate inputs at random to explore app's behavior. There are many tools implementing this kind of strategy; often used to test the robustness of apps.

Monkey[6] 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** [23] 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. **DroidMate** [19] 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.

8.2 Model-Based Strategies

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

DroidBot [22] is one tool implementing this kind of strategy. It uses different methods to dynamically construct a state transition model on-the-fly and consume it to generate test inputs. It works without instrumentation and can therefore be used to examine malware, because malicious apps often check their own signature before triggering malicious behaviors. Users can also integrate their own strategies and use it as a framework. **MobiGUITAR** [2] is a tool that dynamically generates a model of an app during exploration, based on the run-time state of GUI widgets. **SwiftHand** [14] learns a model of the app during testing, consumes this model to generate inputs and uses the execution of these to refine the model.

[6]https://developer.android.com/studio/test/monkey.html

Its key feature is that it avoids restarting the app, which is significantly more expensive than executing a sequence of inputs. It only generates touch and scroll events, no system events. **Stoat** [27] is a tool that uses dynamic and static analysis to reverse-engineer a stochastic model of the app's GUI. During testing, it also randomly injects system events to further enhance the testing effectiveness. It is effective for discovering unique crashes. **ORBIT** [33] also uses a model-based strategy. It uses a grey-box approach for extracting a model of an app. First, it uses static analysis to extract the set of events supported by the app's GUI. Then, it reverse-engineers a model of the app by systematically exercising these events. A^3E **Targeted** [5] is one more model-based tool. It uses a static data flow analysis on the app byte code, to construct an activity transition graph, that captures legal transitions among activities (app screens), and then explores the graph systematically. It directs the exploration to cover all activities - especially activities that would be difficult to reach during normal use.

Our approach is a model-based strategy, which generates an on-the-fly model. In contrast to the aforementioned model-based tools, our approach does not create a state transition model for the app under test, but is targeted at determining how to effectively interact with the app.

8.3 Explorative Strategies

The third main category of exploration strategies are *explorative strategies*.

AndroidRipper [1] is one tool implementing this kind of strategy. It uses a user-interface driven *ripper* to systematically traverse the app's user interface. **IntelliDroid** [32] is a tool that attempts to trigger specific behaviors. It can be configured to produce inputs specific to a dynamic analysis tool, for dynamic malware analysis, and it can determine the precise order these inputs must be injected. **CuriousDroid** [12] can decompose application user interfaces on-the-fly, creating a context-based model that is tailored to the current user layout. It can be used for dynamic sandboxes, which is a widespread approach for detecting malicious applications. A^3E **Depth-First** [5] is a tool that uses the same activity transition graphs as A^3E Targeted. It explores activities and GUI elements in a depth-first manner. It traverses the app in a slower, but more systematic way than A^3E Targeted.

8.4 Reinforcement Learning Strategies

Likewise, the usage of reinforcement learning techniques for automated test generation has been investigated by research.

GUI testing with reinforcement learning [6, 7] showed that reinforcement learning can be used for automated GUI (robustness) testing. As an reinforcement learning algorithm they used *Q-learning* [30], a popular model-free reinforcement learning technique. **Action selection** [17] Esparcia *et al.* 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* can only be achieved through an adequate choice of parameters. **Offline Q-learning** [21] Koroglu *et al.* 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 and 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 the crowd-based exploration by Borges *et al.* [10] than to our approach, as both learn static models. Our approach gains and applies knowledge during testing, generating a model which is specific to the app under test.

9 CONCLUSION AND FUTURE WORK

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. The refined model could test new versions of the same app. This would allow for "continuous learning and testing". It would also be possible to transfer refined models between different apps.

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

Coverage as a reward. Instead of using the effectiveness of an action as a binary reward, one could use *coverage as a reward*, thus leading towards actions which 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:

$$trials_{(a,C(w))} \leftarrow \alpha \times trials + 1 \quad (1)$$

$$wins_{(a,C(w))} \leftarrow \alpha \times wins + r \quad (2)$$

The learning rate adjust whether the algorithm should forget previous results quicker ($\alpha < 1$) and have a downward pressure toward ignorance, or whether the algorithm should act more risky ($\alpha > 1$) 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

ACKNOWLEDGMENTS

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

REFERENCES

[1] 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 *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32, 5 (2015), 53–59.

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

[4] 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.

[5] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Acm Sigplan Notices*, Vol. 48. ACM, 641–660.

[6] Sebastian Bauersfeld and Tanja Vos. 2012. A reinforcement learning approach to automated GUI robustness testing. In *Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE 2012)*. 7–12.

[7] Sebastian Bauersfeld and Tanja EJ Vos. 2014. User interface level testing with TESTAR; what about more sophisticated action specification and selection?. In *SATToSE*. 60–78.

[8] Richard Bellman. 1957. A Markovian decision process. *Journal of Mathematics and Mechanics* (1957), 679–684.

[9] Harald Benzing, Karl Hinderer, and Michael Kolonko. 1984. On the k-armed Bernoulli bandit: Monotonicity of the total reward under an arbitrary prior distribution. *Mathematische Operationsforschung und Statistik. Series Optimization* 15, 4 (1984), 583–595.

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

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

[12] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2016. CuriousDroid: automated user interface interaction for Android application analysis sandboxes. In *International Conference on Financial Cryptography and Data Security*. Springer, 231–249.

[13] Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of Thompson sampling. In *Advances in neural information processing systems*. 2249–2257.

[14] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Acm Sigplan Notices*, Vol. 48. ACM, 623–640.

[15] 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.

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

[17] Anna I Esparcia-Alcázar, Francisco Almenar, M Martínez, Urko Rueda, and T Vos. 2016. Q-learning strategies for action selection in the TESTAR automated testing tool. In *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*. 130–137.

[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 Andreas Zeller. 2016. DroidMate: a robust and extensible test generator for Android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 293–294.

[20] Michael N Katehakis and Arthur F Veinott Jr. 1987. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research* 12, 2 (1987), 262–268.

[21] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of Android applications. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*. IEEE, 105–115.

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for Android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 23–26.

[23] 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*. ACM, 224–234.

[24] Herbert Robbins. 1952. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.* 58, 5 (1952), 527–535.

[25] Julia Rubin, Michael I Gordon, Nguyen Nguyen, and Martin Rinard. 2015. Covert communication in mobile applications (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 647–657.

[26] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. 2018. A tutorial on Thompson sampling. *Foundations and Trends® in Machine Learning* 11, 1 (2018), 1–96.

[27] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256.

[28] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3/4 (1933), 285–294.

[29] Joannes Vermorel and Mehryar Mohri. 2005. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning*. Springer, 437–448.

[30] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[31] Christopher John Cornish Hellaby Watkins. 1989. *Learning from delayed rewards*. Ph.D. Dissertation. King's College, Cambridge.

[32] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.

[33] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.