2018 IEEE 11th International Conference on Software Testing, Verification and Validation

QBE: QLearning-Based Exploration of Android Applications

Yavuz Koroglu and Alper Sen
Department of Computer Engineering
Bogazici University, Istanbul, Turkey
{yavuz.koroglu,alper.sen}@boun.edu.tr

Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi and Yunus Donmez
Netas Telecommunications
Istanbul, Turkey
yunusm@netas.com.tr

***Abstract*—Android applications are used extensively around the world. Many of these applications contain potential crashes. Black-box testing of Android applications has been studied over the last decade to detect these crashes. In this paper, we propose QLearning-Based Exploration (QBE), a fully automated black-box testing methodology, which explores GUI actions using a well-known reinforcement learning technique called QLearning. QBE performs automata learning to obtain a model of the AUT, and generates replayable test suites. Specifically, QBE learns from a set of existing applications the kinds of actions that are most useful in order to reach a particular objective such as detecting crashes or increasing activity coverage. To the best of our knowledge, ours is the first machine learning based approach in Android GUI Testing. We conduct experiments on a test set of 100 AUTs obtained from the commonly used F-Droid benchmarks to show the effectiveness of QBE. We show that QBE performs better than all compared black-box tools in terms of activity coverage and number of distinct detected crashes. We make QBE and our experimental data available online.**

***Keywords*-Android; Test Generation; Reinforcement Learning; Automata Learning**

I. Introduction

Mobile applications have become an essential part of our daily life. Statistics show that an average person uses mobile phones 3 hours a day and spends 90% of this time on mobile applications (non-browsing activity) [1]. Following the growing trend of the mobile application market, there is an increasing focus on mobile application testing in top testing conferences and journals [2]. These testing tools implement white-box, gray-box, or black-box testing techniques, focusing mainly on Android applications.

The advantage of black-box testing for Android applications is that it only requires the binary file and a suitable environment to dynamically execute the Application Under Test (AUT). There are several recent tools on automated black-box testing of Android applications. These tools are PUMA [3], A^{3}E [4], SwiftHand [5], Sapienz [6], CrashScope [7], and DynoDroid [8]. Each tool has its own contribution to the state-of-the-art such as depth-first and targeted exploration strategies in A^{3}E, the usage of cosine similarity for state equivalence in PUMA, approximate model learning through systematic execution in SwiftHand, addition of contextual states in CrashScope, biased random testing of DynoDroid and the Pareto optimal Search-Based Software Engineering approach in Sapienz.

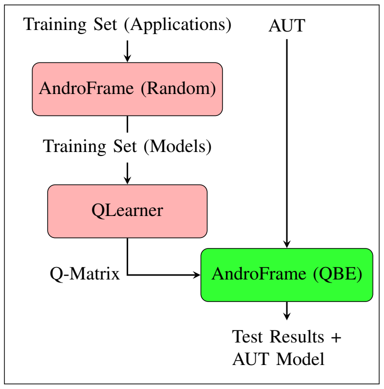


Fig. 1. QLearning-Based Exploration (QBE) Overview

There are also several Model-Based Testing (MBT) tools for Android application testing where MBT assumes that a model of the AUT is available. Tools such as MobiGUITAR [9] and SwiftHand [5] automatically generate a model of the application. For example, SwiftHand employs automata learning methods where GUI actions of the application are traversed in a Depth-First Exploration (DFE) strategy. MBT tools have the advantage of capturing behavioral information of the AUT in a model. This model can be used to generate meaningful test cases for the AUT. Also, the model itself can be tested or verified to avoid the cost involved in testing the AUT.

Despite the ongoing development in the state-of-the-art, a simple black-box random testing tool, Monkey [10], outperforms complex tools in terms of coverage and the number of found crashes [7], [4]. The disadvantage of Monkey is that tests are hard to reproduce and faults are hard to localize. More importantly, Monkey can be used to generate thousands of GUI events per second, which is unrealistic since a human agent will not be able to generate events at that rate. Similarly, Monkey also generates sudden drops or increases in battery level, and sudden changes in the orientation of the phone which results in unrealistically high accelerometer values.

We propose QLearning-Based Exploration (QBE), a fully automated black-box testing methodology that explores GUI actions using a well-known reinforcement learning technique in machine learning, called QLearning [11]. QBE explores GUI actions of the application according to a pre-approximated probability distribution of satisfying an objective. An example objective can be increasing the activity coverage or detecting a crash. We call this probability distribution the transition prioritization matrix (i.e. Q-Matrix) and estimate the transition prioritization matrix using QLearning. Then, during exploration of the AUT we sample new GUI actions from the Q-Matrix rather than using a random or depth-first exploration strategy. For example, QBE may learn that of all enabled actions on a screen the probability of a click action has a better chance of leading to a crash. Given the large state space of the applications, this information is crucial for improving the state-of-the-art in Android application testing.

We show the overview of our approach in Figure 1. We first implement a new fully automated black-box testing tool called AndroFrame. AndroFrame is a modular Android testing framework that includes an online automata learning variant, which obtains models for given AUTs during execution using different strategies. In AndroFrame, we implement Random Exploration (RE) and Depth-First Exploration (DFE) strategies, as well as our novel QLearning-Based Exploration (QBE) strategy. AndroFrame combines features mentioned in previous Android GUI Testing tools; covers all entry points (exportable activities) of the AUT as proposed in A^{3}E [4], obtains a model of the AUT as proposed in SwiftHand [5], uses cosine similarity to distinguish different states of the model as proposed in PUMA [3], and supports contextual states of the AUT as proposed in CrashScope [7]. Furthermore, AndroFrame generates reproducible test suites which is crucial for debugging as well as for obtaining a regression test suite.

We implement QBE on top of AndroFrame. In the training phase of QBE, first, we obtain models of Android applications from a training set by executing them with Random Exploration (RE) strategy. Then, using all these models we obtain a single Q-Matrix with respect to an objective. In the testing phase of QBE, we test a given Application Under Test (AUT) by sampling actions from the distribution implied by the Q-Matrix instead of randomly choosing from the set of all enabled actions. We also obtain a model of the AUT to facilitate Model-Based Testing.

Our main contribution in this paper is QLearning-Based Exploration (QBE) for Android application testing. To the best of our knowledge, ours is the first machine learning based approach in Android GUI Testing. We note that machine learning is different from automata learning and we use machine learning in conjunction with automata learning. For experiments, we randomly select 300 AUTs from F-Droid benchmark suite [12]. We train our QLearning algorithm on 200 AUTs for both crash detection and increasing activity coverage. Using the remaining 100 AUTs as test set, we compare QBE with Monkey, PUMA, SwiftHand, Sapienz, DynoDroid, and Depth-First Exploration (DFE) and Random Exploration (RE) strategies of AndroFrame. We show that on average, QBE achieves the best activity coverage with 78% when it is trained for increasing activity coverage. We also show that on average, QBE finds the highest number of distinct crashes by 13 when it is trained for crash detection.

We organize our paper as follows. In Section II, we describe related testing approaches. We describe Android background in Section III. In Section IV, we describe our automata learning framework. In Section V, we describe our QLearning methodology. In Section VI, we evaluate our tool as well as the state-of-the-art Android testing tools and discuss the results. In Section VII, we discuss important design decisions and threats to validity. We summarize our results in Section VIII.

II. Related Work

In this section, we first describe related studies in Android testing tools. Then, we discuss testing approaches that use QLearning. Finally, we discuss record and replay techniques.

Monkey [10] is the first black-box Android testing tool. It executes thousands of random events per second. Monkey is known to find the highest number of crashes [7] and achieves the highest activity coverage [4] so far. The downside of Monkey is that, in general, tests are not reproducible and there is not a good way to debug crashes.

A^{3}E [4] systematically executes GUI components using depth-first exploration or targeted exploration strategies. The public version of the tool supports depth-first exploration only. A^{3}E's depth-first exploration triggers each widget at least once, whereas our depth-first exploration triggers each widget at each state.

PUMA [3] is another black-box Android testing tool. PUMA's main contribution is the concept of *cosine similarity* which is based on the comparison of contents of two states. AndroFrame uses cosine similarity, as well as the set of enabled actions to decide state equivalence. Hence, our state equivalence relation is the most precise compared to the state-of-the-art.

CrashScope [7] offers user-friendly reports on detected crashes, which increase reproducibility and improve debugging. CrashScope's main contribution is the introduction of *contextual states* during testing such as wifi, GPS, and rotation that may trigger crashes. We also provide support for contextual states in AndroFrame.

Sapienz [6] is a state-of-the-art search-based Android testing tool that uses evolutionary algorithms to generate fault revealing test cases or to minimize test suites.

DynoDroid [8] is a guided random testing tool for Android. DynoDroid supports system events as well as GUI actions. We devise a new exploration strategy (QBE) in AndroFrame different from the random exploration strategy used in DynoDroid. DynoDroid uses instrumentation to deduce *relevant* events to guide the exploration, whereas AndroFrame does not instrument the AUT at all to determine relevant events. Also, our Q-Learning approach can be trained for different objectives, whereas DynoDroid has a fixed objective.

SwiftHand [5] is another black-box Android testing tool that learns an approximate model of the AUT using a modified L^* algorithm. To the best of our knowledge, SwiftHand is the first tool that applies automata learning in Android testing but it does not use machine learning.

We use a reinforcement learning technique in QBE. Reinforcement learning is a semi-supervised machine learning scheme. Reinforcement learning is concerned with how software *agents* should take *actions* in an *environment* such that a cumulative *reward* is maximized. Reinforcement learning differs from standard supervised learning techniques in that correct input/output pairs are never given, but only a reward is observed if the agent reaches an *objective*. Among other reinforcement learning techniques, QLearning is the most appropriate for GUI testing because other techniques such as Monte Carlo and Brute Force require many actions to be executed during learning, which is costly.

AutoBlackTest [13] is the first GUI testing tool that uses QLearning method. The main difference between AutoBlackTest and our method is that AutoBlackTest learns app-specific Q-values, and hence it must be trained for each application. We learn a single matrix of Q-values which we use for all applications, and hence we avoid training for each application during testing. AutoBlackTest uses a reward function which is based on the number of changing widgets between two states to reach an objective using QLearning. Tracking the change of widgets allow them to decide whether the screen has changed or not. We use a similar reward function, called activity coverage that allows us to determine whether an application screen (activity) has changed or not. We also add a new reward function for crash detection, which is especially important for mobile testing.

To the best of our knowledge, ours is the first machine learning based approach in Android GUI Testing.

Another approach that uses QLearning is AntQ [14]. It is a GUI testing methodology which uses ant colony optimization technique. They increase the performance of their optimization by incorporating QLearning in the process. Similar to AutoBlackTest, AntQ learns a new Q-Matrix for each AUT, whereas our QLearning method learns a single Q-Matrix from all AUTs in the training set, which we use on the AUTs in the test set.

VALERA [15], RERAN [16], and BARISTA [17] are generic record and replay tools for Android. AndroFrame has a simpler method that can record and replay only the test cases that are obtained during model generation. Hence, we can reproduce crashes. This is crucial for debugging as well as for obtaining a regression test suite. None of the above Android testing tools except DynoDroid, Sapienz, and A^{3}E provide this functionality.

III. Android Background

We now describe basic features of the Android GUI to facilitate the understanding of our methodology.

Android GUI is based on *activities* and *events*. An *activity* represents a single screen with a user interface and contains GUI components (widgets). Each GUI component (such as button or text box) has properties describing the boundaries of the component in pixels (x_1, y_1, x_2, y_2) and how the user can interact with the component through actions. Example GUI component properties are *enabled*, *clickable*, *longclickable*, *scrollable*, *type*, and *password*.

TABLE I
List of GUI Actions

Non-Contextual	Param1	Param2	Param3	Param4	Param5
click	x	y	-	-	-
longclick	x	y	-	-	-
text	x	y	string	-	-
swipe	x1	y1	x2	y2	duration
menu	-	-	-	-	-
back	-	-	-	-	-

Contextual	Parameter
connectivity	on/off/toggle
bluetooth	on/off/toggle
location	gps/gps&network/off/toggle
planemode	on/off/toggle
doze	on/off/toggle

Special	Param1	Param2	Param3	Param4	Param5
reinitialize	package	activity	-	-	-

The Android system and the user can interact with GUI components using *events*. We divide events in two categories, *system events* and *GUI events (actions)*. We show the list of GUI actions that we use in Table I, which covers more actions then are typically used in the literature. We group actions into three categories; non-contextual, contextual, and special. Non-contextual actions correspond to actions that are triggered by user gestures. *Click* and *longclick* take two parameters, x and y coordinates to click on. *Text* takes three parameters, x and y for coordinates and string to describe what to write. *Swipe* takes five parameters. The first four parameters describe the starting and the ending coordinates. The fifth parameter is used to adjust the speed of *swipe*. *Menu* and *back* actions have no parameters. These actions just click to the menu and back buttons of the mobile device, respectively. Contextual actions correspond to the user changing the contextual state of the AUT. Contextual state is the concatenation of the global attributes of the mobile device (internet connectivity, bluetooth, location, planemode, sleeping). *Connectivity* action adjusts the internet connectivity of the mobile device (adjusts wifi or mobile data). *Bluetooth*, *location*, and *planemode* are straightforward. *Doze* action taps the power button of the mobile device and puts the device to sleep or wakes it. We use *doze* action to pause and resume the AUT. We also use a special action called *reinitialize*, which reinstalls and starts an AUT. System events are system generated events, e.g. *battery level*, *receiving SMS*, and *clock/timer*.

Finally, we define a *crash* in Android as a fatal exception in Android logs. Crashes often result with the AUT terminating with or without any warning. Some crashes do not visually affect the execution, but the screen becomes unresponsive as a result.

IV. AndroFrame Background

In this section, we describe our automata learning black-box test generation framework, AndroFrame. We implement several exploration strategies including random exploration, depth-first exploration, and Q-Learning based exploration in AndroFrame.

We use the *Extended Labeled Transition System* (ELTS) [5] as a model for the AUT. Formally, an ELTS M is a 5-tuple, $M = (V, v_0, Z, \omega, \lambda)$, where

- V is a set of *states* (vertices),
- $v_0 \in V$ is the *initial state*,
- Z is the set of all *actions* (input alphabet),
- $\omega : V \times V \times Z$ is the *transition relation*, and
- $\lambda : V \rightarrow vevenegatif(Z)$ is the *enabling function*, where $\forall v in V, \lambda(v) \subseteq Z$ denotes set of actions enabled at state v.

We define a GUI state, or simply a *state* v to be the concatenation of the following:

1) Package Name (a name representing the AUT)
2) Activity Name
3) Contextual State
4) GUI Components

Each state v has a set of enabled actions $\lambda(v)$, extracted from its set of GUI components. We say that a GUI action, or simply an *action* $z \in \lambda(v)$ *is enabled* at state v iff we can deduce that z interacts with at least one GUI component in v. We define a *transition* as a 3-tuple, (start-state, end-state, action), shortly denoted by (v_s, v_e, z). We define an execution trace, or simply a *trace* t, as a sequence of transitions. An example trace is as follows.

$$t = (v_1, v_2, z_1), (v_2, v_3, z_2), \dots, (v_n, v_{n+1}, z_n)$$

where n is the *length* of the trace.

We say that a trace t is a *test case* if the first state of the trace is the initial state v_0 (the GUI state when the AUT is started). A *test suite* TS is a set of test cases.

Definition 1: An ELTS $M = (V, v_0, Z, \omega, \lambda)$ is *deterministic* iff

$$\forall v, v', v'' \in V, \forall z \in \lambda(v),$$
$$[(v, v', z) \in \omega \text{ and } (v, v'', z) \in \omega] \rightarrow (v' = v'')$$

In other words, each transition in a model M must have one unique resulting state in order to make M deterministic.

We describe how we generate test suites using AndroFrame in Algorithm 1. We take an AUT, a timeout in seconds, and a number denoting the maximum number of actions per test case as input. We start with an empty test suite in line 1. Then, we initialize our model in line 2. We use a timeout in seconds as termination condition in line 3. For each test case, we start with reinitializing the AUT in line 4. Reinitialization reinstalls and restarts the AUT. We generate each test case in the test suite between lines 6-23. We choose one of the enabled actions in line 7. Note that we can use Random Exploration (RE), Depth-First Exploration (DFE) or Q-Learning Based Exploration (QBE) strategy for choosing the action. We execute this action and check if the AUT crashed or not. If the AUT crashed, we add an edge from the previous state to "crash" state in the transition relation in line 10 and add the transition to our test case in line 11. Otherwise; we add the next state to the set of states, add the new actions to the set of actions, add the transition to the transition relation, and add the transition to the test case in lines 14-17. If the model becomes non-deterministic, we call a variant of the PassiveLearn algorithm to make the model deterministic by adding new states. We continue the inner loop until any of the conditions in line 23 becomes true and then move onto the generation of the next test case.

Algorithm 1 AndroFrame Test Suite Generation Algorithm

```
Require:
    $AUT$ : Application Under Test
    $X \in \mathbb{N}$ : Timeout (in seconds) to terminate testing
    $N \in \mathbb{N}$ : Maximum number of actions per test case
Ensure:
    $TS$ : A test suite for the AUT
    $M = (V, v_0, Z, \omega, \lambda)$ : Model of the AUT

 1: $TS \leftarrow \{\}$
 2: $M \leftarrow (\{v_0\}, v_0, \{\}, \{\}, \{\})$
 3: while elapsed time is less than $X$ do
 4:     Execute "reinitialize"
 5:     $v \leftarrow v_0$
 6:     repeat
 7:         $z \leftarrow$ choose $z \in \lambda(v)$
 8:         Execute $z$
 9:         if AUT is crashed then
10:             $\omega \leftarrow \omega \cup \{(v, \text{crash}, z)\}$
11:             $t \leftarrow t \cdot (v, \text{crash}, z)$          ▷ Concatenation
12:         else
13:             $v' \leftarrow$ current state of the AUT
14:             $V \leftarrow V \cup \{v'\}$
15:             $Z \leftarrow Z \cup \lambda(v')$
16:             $\omega \leftarrow \omega \cup \{(v, v', z)\}$
17:             $t \leftarrow t \cdot (v, v', z)$          ▷ Concatenation
18:             if $M$ is non-deterministic then
19:                 $(M, TS) \leftarrow$ PassiveLearn$(M, TS, t)$
20:             end if
21:             $v \leftarrow v'$
22:         end if
23:     until elapsed time is greater than $X$ or length of $t$ is
        larger than $N$ or AUT is crashed
24:     $TS \leftarrow TS \cup \{t\}$
25: end while
```

A. Cosine Similarity for State Equivalence

We require an *equivalence relation* in line 14 of Algorithm 1 during the union operation $V \cup \{v'\}$. We now formally define our state equivalence relation (cosine similarity) between two states v and v' which is based on the definition given in PUMA. We first calculate content vectors of these states denoted by cv and cv', using metrics collected from each corresponding state and then take the cosine similarity of these content vectors. Formally,

$$\forall v, v' \in V, v = v' \text{ iff}$$
$$[\lambda(v) = \lambda(v')] \wedge [N(v) = N(v')] \wedge [\cos(cv, cv') > 0.95] \quad (1)$$

In Equation 1, two states v and v' are equivalent iff they both have the same set of enabled actions, the same number of components, and the cosine similarity of their content vectors cv and cv' is above 0.95, which was determined by PUMA [3]. Note that our state equivalence check sacrifices accuracy for

Algorithm 2 PassiveLearn Algorithm

```
Require:
    $M = (V, v_0, Z, \omega, \lambda)$ : Non-deterministic AUT model
    $TS$ : A test suite for the AUT
    $t = (v_1, v_2, z_1) \dots (v_n, v_{n+1}, z_n)$ : A test case
Ensure:
    $M' = (V', v'_0, Z', \omega', \lambda')$ : Deterministic AUT model
    $TS'$ : Updated Test Suite

1: $(M', TS') \leftarrow (M, TS \cup \{t\})$
2: $v'_n \leftarrow v_n$
3: $v'_n \leftarrow v_n \cdot$ dummy component s.t. $v'_n \neq v_n, \lambda(v'_n) = \lambda(v_n)$
4: $\omega' \leftarrow \omega' - \{(v_n, v_{n+1}, z_n)\}$
5: $\omega' \leftarrow \omega' - \{(v_{n-1}, v_n, z_{n-1})\}$
6: $\omega' \leftarrow \omega' \cup \{(v_{n-1}, v'_n, z_{n-1})\}$
7: for $t' \in TS'$ do
8:     for $i \in \mathbb{Z}^+$ s.t. $i \leq$ length of $t'$ do
9:         if $\exists d \in \mathbb{N}$ s.t. $t'_i = (v_{n-1}, v_n, z_n)$ then
10:            $t'_i \leftarrow (v_{n-1}, v'_n, z_n)$
11:        end if
12:    end for
13: end for
14: for $t' \in TS'$ do
15:    for $i \in \mathbb{Z}^+$ s.t. $i \leq$ length of $t'$ do
16:        $(v'_i, v'_{i+1}, z'_i) \leftarrow t'_i$
17:        $\omega' \leftarrow \omega' \cup \{(v'_i, v'_{i+1}, z'_i)\}$
18:        if $M'$ is non-deterministic then
19:            return PassiveLearn$(M', TS', t'_{1 \dots i})$
20:        end if
21:    end for
22: end for
23: while $\exists v \in V'$ s.t. $\neg\exists(v', z) \in V' - \{v_0\} \times Z', (v', v, z) \in \omega'$ do
24:    $V' \leftarrow V' - \{v\}$
25:    $\omega' \leftarrow \omega' - \{(v, v', z)\}, \forall(v', z) \in V' \times Z'$
26: end while
```

simplicity and performance. Hence, even with no time limit, we may not capture the exact model of the AUT. However, our check suffices for finding bugs or increasing coverage as shown in experiments.

B. Passive Learning Algorithm

We now describe our implementation of PassiveLearn used in line 19 of Algorithm 1 for generating a deterministic model of the AUT. Algorithm 2 for PassiveLearn is based on the algorithm described in Swifthand [5].

In Algorithm 2, we take the non-deterministic model, the test suite, and the test case that causes the non-determinism in the model as input. We know that PassiveLearn is called when the last generated transition in Algorithm 1 causes non-determinism. This non-determinism occurs because our definition of state is not 100% accurate and does not capture the complete device state. To fix non-determinism, we first duplicate the problematic state as v'_n in line 2. Then we modify v'_n by adding a dummy GUI component to it so that it is

Algorithm 3 QLearning-Based Exploration (QBE)

```
Require:
    $M = (V, v_0, Z, \omega, \lambda)$ : Current ELTS of the AUT
    $v_c \in V$ : Current State of the AUT
    $\beta : V \to S$ : State Labeling Function
    $\alpha : Z \to \Sigma$ : Action Labeling Function
    $\vec{Q_o} \in [0,1]^{S \times \Sigma}$ : Transition Prioritization Matrix
Ensure:
    $z \in \lambda(v_c)$ : An action

1: $qMap \leftarrow \{\}$                          ▷ $qMap : \Sigma \to [0,1]$
2: for all $z \in \lambda(v_c)$ do
3:     $qMap \leftarrow qMap \cup (\alpha(z), \vec{Q_o}[\beta(v_c), \alpha(z)])$
4: end for
5: $a \leftarrow$ random $a$
       with probability proportional to $q$
       s.t. $(a, q) \in qMap$
6: return random $z \in \lambda(v_c)$ s.t. $[a = \alpha(z)]$
```

no longer equivalent to v_n. Note that the properties of the dummy state is irrelevant, as long as it has exactly the same enabled actions with the original state. We also remove the last two transitions added to the transition relation from the model in lines 4 and 5. Instead, we add a new transition to the transition relation which is going to v'_n in line 6. We replace all transitions in all test cases that were going from v_{n-1} to v_n with transitions going from v_{n-1} to v'_n in lines 7-13. Then, we add any missing transitions to the transition relation in lines 14-22. However, this process may make the model non-deterministic again. In that case, we recursively call the PassiveLearn algorithm to correct the model once more. Finally, we remove all states and their outgoing transitions from the model that are disconnected from the model in lines 23-26.

V. QLearning-Based Exploration (QBE) Methodology

In this section, we define our novel exploration strategy, QLearning-Based Exploration (QBE). Then, we describe how to estimate the input of QBE, called transition prioritization matrix using QLearning reinforcement technique.

A. QLearning-Based Exploration (QBE)

QBE explores GUI actions of the application according to a pre-approximated probability distribution of satisfying an objective. During exploration of the AUT in line 7 of Algorithm 1 instead of for example randomly picking an action, we sample new GUI actions from the Q-Matrix. QBE can use the knowledge that the probability of clicking on a screen with several enabled actions leads to a crash. We first formally define several terms in order to explain QBE then we describe the QBE algorithm.

An *objective* o is a function $o : V \times Z \to \{0, 1\}$, where V is the set of states and Z is the set of actions. In other words, we divide all state-action pairs into two groups, the ones that satisfy the objective and the ones that do not. Example objectives can be *increasing activity coverage* or *detecting a crash*. A trace t *satisfies* an objective o iff the last state-action pair in the trace satisfies the objective.

In order to define *trace similarity*, we define a set of *abstract states*, S and a set of *abstract actions*, Σ. An example set of *abstract states* S can be formed by categorizing states by the number of enabled actions in that state, such as

$$S = \{\textit{too-few}, \textit{few}, \textit{moderate}, \textit{many}, \textit{too-many}\}$$

An example set of *abstract actions* Σ is

$$\Sigma = \{\textit{menu}, \textit{back}, \textit{click}, \textit{longclick}, \textit{text}, \textit{swipe}, \textit{contextual}\}$$

We define an *abstract trace* $p = \rho(t)$ of a trace t as a sequence of abstract state action pairs. An abstract state-action pair $(s, a) \in S \times \Sigma$ is obtained from a state-action pair $(v, z) \in V \times Z$ using two labeling functions $\beta : V \to S$ and $\alpha : Z \to \Sigma$.

We say that two abstract traces p and p' are *equal* iff they have the same abstract state-action pairs in the same order. We say that traces t and t' are *similar* iff their abstract traces $p = \rho(t)$ and $p' = \rho(t')$ are equal.

We define the probability of an abstract trace p satisfying an objective o as the number of traces t such that $p = \rho(t) \wedge t$ satisfies o divided by the number of all traces such that $p = \rho(t)$.

Definition 2: A *transition prioritization matrix (Q-Matrix)* is a 2D matrix $\vec{Q_o} \in [0,1]^{S \times \Sigma}$, where S is the set of abstract states and Σ is the set of abstract actions and each cell of $\vec{Q_o}$ is an estimate of the probability of a distinct abstract state-action pair (s, a) being an element of a abstract trace $p = \rho(t)$ which has a high probability of satisfying o.

Informally, an abstract state-action pair has a high value in the transition prioritization matrix if the corresponding concrete state-action pairs of the abstract state-action pair have high probability of being an element of a trace t which satisfies the objective function o. Note that multiple state-action pairs can correspond to the same abstract state-action pair as well as multiple traces can correspond to the same abstract trace.

In Algorithm 3, we present our novel exploration strategy, QLearning-Based Exploration (QBE), which takes five inputs; the model M, the current state v_c, the state labeling function β, the action labeling function α, and the transition prioritization matrix $\vec{Q_o}$. The goal of the algorithm is to return one of the enabled actions at the current state by using the transition prioritization matrix. The rows of the matrix describe the abstract states whereas the columns describe the abstract actions. Each cell has a value between 0 and 1. We obtain the abstract state for the current state using the state labeling function. In line 3, we add probabilities of all enabled abstract actions of the abstract state to a map. Then, we choose one enabled abstract action from this map where the choice is proportional to the probabilities existing in the map. This step is described in line 5. Then, in line 6, we randomly return one of the concrete actions of the abstract state found in previous step. Note that when QBE is used in Algorithm 1, α, β, and $\vec{Q_o}$ should be given as constants.

Algorithm 4 QLearning Algorithm

```
Require:
    $o : V \times Z \to \{0,1\}$ : Objective
    $MS =$ set of all $M = (V, v_0, Z, \omega, \lambda)$ : ELTS of all AUTs
    $\beta : V \to S$ : State Labeling Function
    $\alpha : Z \to \Sigma$ : Action Labeling Function
    $\epsilon_u \in [0,1]$ : Epsilon Update Factor
    $L \in \mathbb{Z}^+$ : Maximum Number of Actions
    $\gamma \in [0,1]$ : Discount Factor
Ensure:
    $\vec{Q} \in [0,1]^{|S| \times |\Sigma|}$ : Transition Prioritization Matrix

 1: $\vec{Q} \leftarrow 0, \vec{N} \leftarrow 0, \epsilon \leftarrow 1$
 2: for all $M \in MS$ do
 3:     $j \leftarrow 0$
 4:     $v \leftarrow v_0$
 5:     $rnd \leftarrow U[0,1]$                       ▷ Generate uniform random
 6:     if $rnd < \epsilon$ then
 7:         $z \leftarrow$ random $z \in \lambda(v)$
 8:     else
 9:         $z \leftarrow z \in \lambda(v)$ s.t.
            $\max_a \vec{Q}[\beta(v), a] = \vec{Q}[\beta(v), \alpha(z)]$
10:     end if
11:     repeat
12:         $v' \leftarrow v' \in V$ s.t $(v, v', z) \in \omega$
13:         $rnd \leftarrow U[0,1]$                   ▷ Generate uniform random
14:         if $rnd < \epsilon$ then
15:             $z' \leftarrow$ random $z' \in \lambda(v')$
16:         else
17:             $z' \leftarrow z' \in \lambda(v')$ s.t.
                $\max_a \vec{Q}[\beta(v'), a] = \vec{Q}[\beta(v'), \alpha(z')]$
18:         end if
19:         $s \leftarrow \beta(v), s' \leftarrow \beta(v'), a \leftarrow \alpha(z), a' \leftarrow \alpha(z')$
20:         $\vec{N}[s,a] \leftarrow \vec{N}[s,a] + 1$           ▷ Update history
21:         $\vec{Q}[s,a] \leftarrow \vec{N}[s,a]^{-1} \left(o(v,z) + \gamma \vec{Q}[s',a'] - \vec{Q}[s,a]\right)$
    $+ \vec{Q}[s,a]$                                  ▷ Update $\vec{Q}$
22:         $\vec{Q}[s] \leftarrow \vec{Q}[s] / \sum_a \vec{Q}[s,a]$   ▷ Normalize over rows
23:         $v \leftarrow v', z \leftarrow z', j \leftarrow j + 1$
24:     until $(j \geq L) \vee v = crash \vee v$ is terminal
25:     $\epsilon' \leftarrow \epsilon \cdot \epsilon_u$                ▷ Multiply the learning factor
26: end for
27: return $\vec{Q}$
```

B. Estimating the Transition Prioritization Matrix

We now show how to use a reinforcement learning technique called QLearning to estimate the transition prioritization matrix $\vec{Q_o}$ used in Algorithm 3. This step occurs before the exploration starts in an offline fashion.

In Machine Learning, finding shortest paths from the initial state to a goal state in an unknown terrain or more is solved via reinforcement learning [18]. One of the most common reinforcement learning techniques is QLearning [11], which has previously been used in GUI testing [13] as we described earlier.

We formally present QLearning iteratively in Algorithm 4.

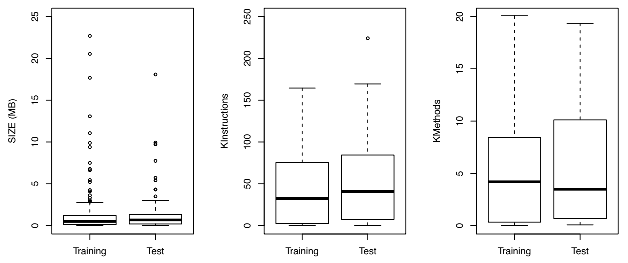


Fig. 2. AUT Characteristics of Training Set, Test Set and F-Droid Benchmarks

In line 2, we run the algorithm for each AUT one by one and obtain a single transition prioritization matrix that includes the contributions of all AUTs. The order of selecting models of AUTs is random, and leads to the same Q-Matrix even after five different runs of the algorithm. Initially, we start with $\vec{Q}$ as all zeros denoting that we have no apriori knowledge that any abstract state-action pair would have a higher chance of satisfying the objective o. The history matrix $\vec{N}$ holds a running count of occurrences of each abstract state-action pair (s, a). Initially, $\vec{N}$ is all zeros. Based on the comparison of a random variable rnd and ϵ (the *learning factor*) in lines 6 and 14, we either randomly choose the next action as shown in lines 7 and 15 or we pick the next action based on the transition prioritization matrix $\vec{Q}$ we learned so far in lines 9 and 17. Initially, we start with $\epsilon = 1$, which denotes that QLearning should do a fully random exploration for the first AUT. At the end of each iteration, we update ϵ by multiplying it with a constant $\epsilon_u \in [0, 1]$ in line 25. We limit the number of actions in an iteration by a constant $L \in \mathbb{Z}^+$ as seen in line 24. We update the history matrix $\vec{N}$ in line 20. We use a constant $\gamma \in [0, 1]$ as a discount factor in line 21 to decrease the Q values as we get farther away from the objective. At each iteration of QLearning after the first iteration, we use the $\vec{Q}$, $\vec{N}$, and ϵ returned by the previous iteration while keeping the other inputs the same. We picked the constants as $\gamma = 0.9$ and $L = 10$ by trial and error. We picked $\epsilon_u = 0.995$ so that the algorithm will have a small but nonzero ϵ even after it is trained for all AUTs.

VI. Evaluation

In this section, we evaluate our new exploration strategy QBE when trained for increasing activity coverage (QBEa) and when trained for detecting crashes (QBEc) by answering two research questions:

RQ1: Activity Coverage: What is the performance of QBEa compared to other black-box Android testing tools in terms of activity coverage?

RQ2: Crash Detection: What is the performance of QBEc compared to other black-box Android testing tools in terms of detection of distinct crashes?

For evaluation, we implement Algorithms 1-4, as well as the Depth-First and Random Exploration strategies in AndroFrame, which is available online for reproducibility [19].

A. Experimental Environment

We performed experiments on an Intel x86 machine with 1TB harddisk, 8x1.6 GHz CPUs containing 8MB L3 cache and running Ubuntu 12.04 operating system. We installed A^{3}E, Dynodroid, PUMA, SwiftHand, and Sapienz as the state-of-the-art for Android testing tools. We use Android SDK version 25.2.4 and an Android 4.4.r5 x86 image on VirtualBox, since this configuration is compatible with most of the testing tools. The publicly available versions of Sapienz and Dynodroid are designed to work with the standard Android Emulator [20] and not the VirtualBox image. Hence, we used the Android Emulator to execute Sapienz and Dynodroid.

We downloaded a total of 300 random AUTs from F-Droid benchmark suite [12]. We formed a training set of 200 AUTs out of 300 with random selection. Then, we formed our test set using the remaining 100 AUTs. We compare the characteristics of our training and test sets in Figure 2. Box plots show that both the training set and the test set have similar characteristics in terms of application size (in megabytes), number of instructions (in thousands), and number of methods (in thousands).

We trained QBE using two objectives, *increasing activity coverage* and *crash detection* on ELTS models of the training set. We obtained ELTS models by executing Random Exploration (RE) strategy of AndroFrame for 10 minutes for each AUT in training set.

Our transition prioritization matrix (Q-Matrix) learning process includes randomness and our results may be affected by this randomness. Hence, we reexecuted the same training process 5 times in order to verify that we obtain the same Q-Matrix each time. In order to obtain the Q-Matrix, we divide states into 5 abstract states according to the number of enabled actions in the state using the functions on left side of Equation 2 shown below. We propose these abstract states by inspecting the mean and variance of the states that we encounter while executing RE. Similarly, we divide actions into 7 abstract actions as in Equation 2. We abstract the details of the actions and group them together. For example, different actions such as clearing a text and writing a very long text have the same abstract action denoted as *text*. Note that we use these abstraction functions as examples, and other functions can be used as well.

$$\beta(v) = \begin{cases} 1, & |\lambda(v)| \leq 1 \\ 2, & |\lambda(v)| \leq 3 \\ 3, & |\lambda(v)| \leq 8 \\ 4, & |\lambda(v)| \leq 15 \\ 5, & |\lambda(v)| > 15 \end{cases} \qquad \alpha(z) = \begin{cases} 1, & z \text{ is a } \textit{menu} \\ 2, & z \text{ is a } \textit{back} \\ 3, & z \text{ is a } \textit{click} \\ 4, & z \text{ is a } \textit{longclick} \\ 5, & z \text{ is a } \textit{text} \\ 6, & z \text{ is a } \textit{swipe} \\ 7, & z \text{ is a } \textit{contextual} \end{cases} \tag{2}$$

Based on abstraction functions in Equation 2, our *transition prioritization matrix*, $\vec{Q}$ is a 5 by 7 matrix. We present our matrices for *increasing activity coverage* ($\vec{Q_a}$) and *crash detection* ($\vec{Q_c}$) in Equations 3 and 4. Some values of these matrices are interpretable such as longclicking action does not contribute to activity coverage or crash detection, since the fourth column is all zeros for both equations. Similarly, we learn that clicking the menu action (hardware menu button) when there are more than 15 enabled actions on a screen

TABLE II
EXPERIMENTAL RESULTS TO ANSWER RQ1 AND RQ2

	RQ1: Coverage		RQ2: Crash
Tool	Activity	Instr.	# Crashes
DFE	63	34	3
RE	58	30	3.2
QBEa	**78**	40	7.8
QBEc	65	32	**12.6**
A^3E	41	17	8
DynoDroid	50	35	5.2
Monkey	60	30	9
PUMA	64	32	6
Sapienz	76	**44**	4
SwiftHand	40	19	0

contributes more to crash detection than to activity coverage, since the probabilities are 0.33 in $\vec{Q_c}$ and 0.06 in $\vec{Q_a}$.

$$\vec{Q_a} = \begin{bmatrix} 0.11 & 0.09 & 0.40 & 0 & 0.10 & 0.30 & 0 \\ 0.13 & 0.44 & 0.26 & 0 & 0.12 & 0.05 & 0 \\ 0.06 & 0.66 & 0.16 & 0 & 0.13 & 0 & 0 \\ 0.17 & 0.25 & 0.40 & 0 & 0.09 & 0.09 & 0 \\ 0.06 & 0.28 & 0.52 & 0 & 0.09 & 0.05 & 0 \end{bmatrix} \tag{3}$$

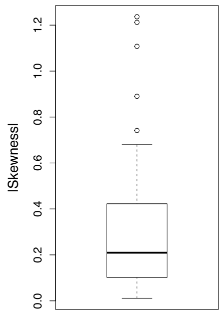
$$\vec{Q_c} = \begin{bmatrix} 0.04 & 0.18 & 0.33 & 0 & 0.12 & 0.33 & 0 \\ 0.19 & 0.18 & 0.12 & 0 & 0.44 & 0.07 & 0 \\ 0.13 & 0.43 & 0.15 & 0 & 0.07 & 0.23 & 0 \\ 0.17 & 0.18 & 0.48 & 0 & 0.18 & 0 & 0 \\ 0.33 & 0.26 & 0.13 & 0 & 0.23 & 0.04 & 0 \end{bmatrix} \tag{4}$$

We execute all testing tools for 10 minutes for each AUT in the test set. We also repeat each execution 5 times, since some degree of randomness is involved with our experimental process due to using probabilities in Q-Matrices described above. We evaluate all tools in terms of activity coverage and crash detection. We also execute AndroFrame using configurations Depth-First Exploration (DFE), Random Exploration (RE), QLearning-Based Exploration with $\vec{Q_a}$ (QBEa), and QLearning-Based Exploration with $\vec{Q_c}$ (QBEc).

B. RQ1: Activity Coverage

We collect activity and instruction coverages using ELLA [21], a binary instrumentation tool for Android applications. Note that ELLA normally only reports method coverage so we extended it to obtain other coverages. We count the covered activities by counting the covered classes that inherit `android.app.activity` class. On top of the covered units (instructions or activities), we also need to know the total unit count of the application to calculate coverage. We statically extract all classes and instructions from the application binary using the Redexer tool [22]. We extract the activities using the same tool, by looking at the subset of all classes of the application that inherit `android.app.activity` class.

Specifically, we first instrument all AUTs of the test set using our modified ELLA tool. We collect instruction and activity coverage of all 100 applications in our test set during execution of each tool. Since we execute each tool 5 times, for each AUT we obtain 500 coverage measurements, we



x_i: Instruction count of activity i
n: Activity count of an AUT
$\lambda = 0.5$: Box-Cox Parameter

$$y_i = (x_i^\lambda - 1)/\lambda,\ \bar{y} = \frac{1}{n}\sum y_i,$$

$$|\text{skewness}| = \left| \frac{\frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^3}{\left[\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2\right]^{\frac{3}{2}}} \right|$$

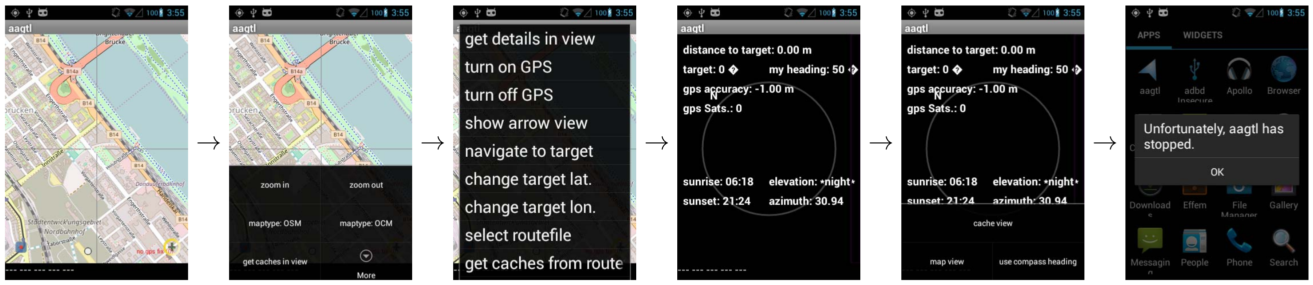
Fig. 3. Box Plot of Skewness of the Test Set and Skewness Calculation

report the average of these measurements in Table II. Table II shows that QBEa and Sapienz achieve higher coverage than other tools. QBEa achieves the best activity coverage, whereas Sapienz is better in instruction coverage.

We perform statistical tests to determine if activity coverage difference between QBEa and Sapienz is statistically significant. We have five measurements for each tool, where Sapienz consistently had the same average activity coverage (76%), and QBEa had noisy activity coverage results (77%, 78%, 79%, 79%, 78%). Since our sample size is small, we use Shapiro-Wilk test for normality. Shapiro-Wilk test can not reject the hypothesis that QBEa results have a normal distribution with p-value = 0.314 and 95% confidence. Then, we perform a one-sided t-test on QBEa results. The results show that we reject the null hypothesis of QBEa results coming from a distribution with a mean less than 76% with p-value = 0.002 and 95% confidence. Power analysis shows that there is only a 0.003 probability of the claim that QBEa has a larger mean than Sapienz is false. Since the tests show that QBEa is significantly better than Sapienz (the tool with the closest activity coverage to QBEa), we conclude that QBEa is also significantly better than every other tool in our experiment without any further statistical tests.

We investigate AUTs where QBEa has better activity coverage than Sapienz. QBEa has better activity coverage in 31% of the applications, Sapienz has better activity coverage in 29% of the applications, and both tools have the same coverage in 40% of the applications. The size, instruction count, and method count of applications where one tool performs better than the other are similar to each other.

We investigate AUTs where Sapienz has higher instruction coverage than QBEa. We look at the evenness of distribution of code over activities of each AUT in our test set. We first apply Box-Cox transformation to our sample distribution, because we can not assume normality. Then, we calculate the sample skewness of this transformed distribution by dividing the third moment of the transformed distribution over the cube of sample standard deviation. We take the absolute value of the skewness. We show our calculations and the box plot of the skewness in Figure 3. If skewness is far from zero, it means that the code is distributed to activities unevenly. Figure 3 shows that 5 applications have unusually high skewness. Our investigation of these applications shows that in all of these applications, Sapienz has better instruction coverage. On



distance to target: 0.00 m
target: 0 my heading: 50
gps accuracy: -1.00 m
gps Sats.: 0
sunrise: 06:18 elevation: *night*
sunset: 21:24 azimuth: 30.94

Unfortunately, aagtl has stopped.

Fig. 4. A Crashing Test Case of *aagtl* Application

average, Sapienz achieves 44% instruction coverage, whereas QBEa achieves 29% coverage in these 5 applications. This is because evolutionary algorithm of Sapienz is optimized to increase instruction coverage, whereas QBEa is trained to increase activity coverage.

We describe an example case where QBEa reaches more activities, *cz.hejl.chesswalk*, a chess engine application. This application has 10 activities, where QBEa reaches 9 and Sapienz reaches 8 activities. The ninth activity that QBE explores is a settings activity. Other testing tools fail to find this activity. Exploration methodologies implemented in RE, PUMA, Sapienz, Dynodroid, and Monkey can not reach the settings activity in the given time, because reachability of the settings activity requires a specific sequence of transitions to be executed, which is hard to hit by random exploration or genetic algorithms. Systematic exploration techniques implemented in DFE and SwiftHand also fail to reach these activities, since they have to exhaust many other sequences before reaching these activities. Similarly, while QBEa is focused on getting to a specific settings activity, Sapienz plays the game more. Hence, since most of the instructions concentrate on the game activity, Sapienz achieves higher instruction coverage. This example leads us to believe that the skewness of distribution of instructions over activities of an application is a possible important characteristic for directing test execution.

Figure II also shows that QBEc does not achieve as much coverage as QBEa. Although a large difference between QBEa and QBEc can be seen in activity coverage, the difference in instruction coverage is small. Hence, we strongly believe that training the QLearning algorithm for increasing activity coverage is crucial to achieve high activity coverage. RE coverage results are similar to Monkey results. This shows us that our testing framework has no significant flaws compared to Monkey, which also performs random exploration. A^{3}E and SwiftHand have the worst coverage. We believe one reason for this is because the number of actions supported by A^{3}E and SwiftHand is small compared to other tools.

Overall, QBE achieves the highest activity coverage compared to other tools, given that the QLearning algorithm is trained for increasing activity coverage.

C. RQ2: Crash Detection

In this case our goal is to count distinct crashes detected by each tool. For this purpose, we periodically check the LogCat tool for fatal exceptions during execution. We count all unique stack traces reported by all fatal exceptions. We call this count the number of distinct crashes detected by a tool. We execute each tool 5 times. Hence, we report means of distinct crashes for each tool in Table II. We observe that QBEc detects 12.6 crashes, followed by Monkey with 9 crashes and A^{3}E with 8 crashes on average. QBEa detects 7.8 crashes on average which shows that it is also crucial to train the QLearning algorithm for crash detection to maximize the number of detected crashes. A^{3}E is able to detect a high number of crashes, despite its poor performance in coverage which shows that high number of crashes is not necessarily correlated with high coverage. Also, SwiftHand was not able to detect any crashes in the given timeout.

We perform statistical tests to determine if crash detection difference between QBEc and Monkey is statistically significant. Both tools have noisy results, so we perform Shapiro-Wilk test for both. We can not reject the normality of Monkey's results (p-value = 0.3254) with 95% confidence. However, we reject the normality of QBEc's results (p-value = 0.046). Hence, we performed a two-sample Wilcoxon test on the results. The test shows that we reject the null hypothesis that QBEc finds fewer crashes on average (p-value = 0.005). Since the tests show that QBEc is significantly better than Monkey (the tool with the closest number of crashes detected), we conclude that QBEc is also significantly better than every other tool in our experiment without any further statistical tests.

We investigate AUTs where QBEc has better crash detection than Monkey. QBEc has better crash detection in 12% of the applications and Monkey has better crash detection in 8% of the applications. The size, instruction count, and method count of applications where one tool performs better than the other are similar to each other.

We investigate a crashing test case for *aagtl* application generated by QBEc within 10 minutes in Figure 4. First, we execute a *menu* action to open up the bottom pop-up menu. Then, we click on *More* button to the bottom-right of the screen. Then, we click on *show arrow view* from the list. Now, a black screen with a circle on it appears. From here, we again execute a *menu* action and click on *cache view* button. Only after these operations, *aagtl* crashes. None of the other tools (PUMA, SwiftHand, Sapienz, Monkey, DynoDroid), including our other methodologies (QBEa, RE, DFE) could detect this crash in 10 minutes. QBEc has a higher chance of finding this crash because it gives higher priority to the *menu* action when there are lots of enabled actions on a screen as shown in Equation 4.

Overall, QBE detects the largest number of distinct crashes compared to other tools, given that the QLearning algorithm is trained for crash detection.

VII. Discussion

In this section, we discuss several design issues and threats to validity. We describe weaknesses and reasons behind our assumptions, design decisions, and experimental procedure.

A. Design Issues

We collect coverage using a a binary instrumentation tool ELLA. This tool does not require the source code of the application and also it can instrument large applications whereas a coverage tool that uses source code instrumentation such as EMMA cannot handle large applications [23]. We also statically collect the total number of activities from binary .dex files using Redexer tool, hence we can find even those activities that are not specified in AndroidManifest.xml.

In Android GUI testing, activities are the main interfaces presented to an end-user. Hence, activity coverage is a commonly used coverage metric in Android GUI testing [4]. We note that GUI testing is orthogonal to unit testing which aims to increase coverage at lower levels. That is, increasing activity coverage and increasing code coverage are distinct training objectives as expected.

Similar to DynoDroid [8], we also define crashes as fatal exceptions and use this as a measure of fair crash comparison. These exceptions correspond to the most severe faults in Android. Also, we report the number of distinct crashes as a measure of crash detection performance, which is a common measure used in Sapienz [6] and DynoDroid [8].

We train our QLearning algorithm by using simple and intuitive abstraction functions since using all properties of a state would lead to a huge representation. It may be possible to improve coverage and crash detection by using better abstraction functions. In future, we plan to develop and test sensitivity of results to our abstraction functions.

QBE assumes that there exists a general probability distribution which applies to all AUTs. Hence, we perform *offline* QLearning which comes up with a fixed probability distribution. Thanks to *offline* QLearning we do not require additional runtime for updating the distribution during testing. It may be argued that different AUTs may require very different sequence of actions to satisfy the same objective (e.g. crash detection or increasing activity coverage) and therefore the general probability distribution may not be enough to explain the patterns for many crashes and activities. In the future, we plan to use *online* QLearning to also account for the AUT-specific patterns.

In AndroFrame, we only consider well-behaving actions (e.g. valid text inputs which we obtain from a manually created dictionary). We will research new techniques to exploit bad-behaving actions to improve crash detection as a future work.

B. Threats to Validity

Internal Validity of our observations hold, since the number of crashes and the coverage measurements can only be affected by the testing algorithm in our experimental environment. We use the same methodology to measure crashes and activity coverage for all testing tools, which makes a fair comparison of testing tools. To increase fairness in our comparisons, we forced Monkey to generate one event in two seconds as also suggested in previous work [7], since this is the rate that other testing tools generate events on average. Note that we keep only the time budget fixed, not the event budget. Event budget of Monkey is constant as a consequence of fixing the delay between events and the total time of testing. F-Droid benchmarks are commonly used in Android GUI Testing studies [24], therefore our test and training sets are not prone to selection bias.

External Validity of our observations also hold. We chose our benchmarks randomly from F-Droid benchmark suite. Randomness, diversity, and the high number of our benchmarks suggest that the experimental results on these benchmarks are externally generalizable. Our benchmark set is diverse with many applications from various domains including news, entertainment, and contact book applications.

Construct Validity of our observations also hold. We take screenshots of each activity during testing to verify the activity is indeed covered. AndroFrame produces replayable test cases, hence we verify crashes via replaying the test cases. We were not able to automatically verify the crashes detected by Monkey, PUMA, and SwiftHand, because they do not readily produce replayable test cases.

VIII. Conclusions and Future Work

In this paper, we proposed QLearning-Based Exploration (QBE), a fully automated black-box testing methodology that explores GUI actions using a well-known reinforcement learning technique in machine learning, called QLearning. Specifically, QBE learns from a set of existing applications the kinds of actions that are most useful in order to reach a particular objective such as detecting crashes or increasing activity coverage. We implement QBE on top of AndroFrame, which also performs automata learning to obtain a model of the AUT and generates replayable test suites. We conducted experiments on 100 applications from F-Droid benchmarks to show the effectiveness of QBE. QBE performed better than all compared black-box tools in terms of activity coverage and number of distinct crashes detected.

As future work, we are going to improve our abstraction functions for states and actions. We will direct our efforts to implement *online* QLearning to increase the number of detected crashes and coverage of our tool. We will also research techniques to exploit bad-behaving actions in test case generation to improve crash detection.

IX. Acknowledgement

This research was supported in part by Bogazici University Research Fund 13662.

References

[1] D. Chaffey, "Statistics on consumer mobile usage and adoption to inform your mobile marketing strategy mobile site design and app development," 2017, http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/.

[2] S. Zein, N. Salleh, and J. Grundy, "A Systematic Mapping Study of Mobile Application Testing Techniques," *J. Syst. Softw.*, vol. 117, pp. 334–356, 2016.

[3] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps," in *12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014, pp. 204–217.

[4] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013, pp. 641–660.

[5] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013, pp. 623–640.

[6] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective Automated Testing for Android Applications," in *25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 94–105.

[7] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 33–44.

[8] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 224–234.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[10] "Android UI/application exerciser monkey," http://developer.android.com/tools/help/monkey.html.

[11] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, 1989, Available at https://www.cs.rhul.ac.uk/home/chrisw/thesis.html.

[12] C. Gultnieks, "F-Droid Benchmarks," 2010, https://f-droid.org/.

[13] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications," in *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[14] S. Carino and J. H. Andrews, "Dynamically Testing GUIs Using Ant Colony Optimization," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 135–148.

[15] Y. Hu and I. Neamtiu, "VALERA: An Effective and Efficient Record-and-replay Tool for Android," in *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2016.

[16] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and Touch-sensitive Record and Replay for Android," in *International Conference on Software Engineering (ICSE)*, 2013.

[17] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[18] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.

[19] "AndroFrame GitHub Page," 2017, https://github.com/yavuzkoroglu/androframe_release.

[20] "The Android Emulator," https://developer.android.com/studio/run/emulator.html.

[21] "ELLA: A Tool for Binary Instrumentation of Android Apps," https://github.com/saswatanand/ella.

[22] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications," in *Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[23] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case," in *39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017.

[24] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" in *30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2015, pp. 429–440.