European Research Council
Established by the European Commission

Self-assessment Oracles for Anticipatory Testing

TECHNICAL REPORT: TR-Precrime-2022-02

Andrea Romdhana[a,b], Alessio Merlo[a], Mariano Ceccato[c], Paolo Tonella[d]
[a]DIBRIS - Università degli Studi di Genova, [b]FBK-ICT, Security & Trust Unit, [c]Università di Verona, [d]Università della Svizzera italiana

Deep Reinforcement Learning for Black-Box Testing of Android Apps

Project no.:	787703
Funding scheme:	ERC-2017-ADG
Start date of the project:	January 1, 2019
Duration:	60 months
Technical report num.:	TR-Precrime-2022-02
Date:	February, 2022
Organization:	Università della Svizzera italiana
Authors:	Andrea Romdhana[a,b], Alessio Merlo[a], Mariano Ceccato[c], Paolo Tonella[d] [a]DIBRIS - Università degli Studi di Genova, [b]FBK-ICT, Security & Trust Unit, [c]Università di Verona, [d]Università della Svizzera italiana
Dissemination level:	Public
Revision:	1.0

Disclaimer:
This Technical Report is a pre-print of the following publication:

Andrea Romdhana, Alessio Merlo, Mariano Ceccato, Paolo Tonella: *Deep Reinforcement Learning for Black-Box Testing of Android Apps*. ACM Transaction on Software Engineering and Methodology (to appear)

Please, refer to the published version when citing this work.

Università della Svizzera Italiana (USI)

Principal investigator: Prof. Paolo Tonella
E-mail: paolo.tonella@usi.ch
Address: Via Buffi, 13 – 6900 Lugano – Switzerland
Tel: +41 58 666 4848
Project website: https://www.pre-crime.eu/

Abstract

The state space of Android apps is huge, and its thorough exploration during testing remains a significant challenge. The best exploration strategy is highly dependent on the features of the app under test. Reinforcement Learning (RL) is a machine learning technique that learns the optimal strategy to solve a task by trial and error, guided by positive or negative reward, rather than explicit supervision. Deep RL is a recent extension of RL that takes advantage of the learning capabilities of neural networks. Such capabilities make Deep RL suitable for complex exploration spaces such as one of Android apps. However, state-of-the-art, publicly available tools only support basic, Tabular RL. We have developed ARES, a Deep RL approach for black-box testing of Android apps. Experimental results show that it achieves higher coverage and fault revelation than the baselines, including state-of-the-art tools, such as TimeMachine and Q-Testing. We also investigated the reasons behind such performance qualitatively, and we have identified the key features of Android apps that make Deep RL particularly effective on them to be the presence of chained and blocking activities. Moreover, we have developed FATE to fine-tune the hyperparameters of Deep RL algorithms on simulated apps, since it is computationally expensive to fine-tune them on real apps.

Contents

1 Introduction

The complexity of mobile applications (hereafter, apps) keeps growing, as apps always provide more advanced services to the users. Nonetheless, it is of utmost importance that they work properly once they are published on app markets, as most of their success/failure depends on the user's evaluation. Therefore, an effective testing phase is fundamental to minimize the likelihood of app failures during execution.

However, automated testing of mobile apps is still an open problem, and the complexity of current apps makes their exploration trickier than in the past, as they can contain states that are difficult to reach and events that are hard to trigger.

There exist several approaches to automated testing of mobile apps that aim to maximize code coverage and bug detection during testing. Random testing strategies [22, 35] stimulate the App Under Test (AUT) by producing pseudo-random events. However, random exploration with no guidance may get stuck when dealing with complex transitions. Model-Based strategies [5, 45, 23] extract test cases from navigation models built employing static or dynamic analysis. If the model accurately reflects the AUT, a Deep exploration can be achieved. Nonetheless, automatically constructed models tend to be incomplete and inaccurate. Structural strategies [7, 17, 36] generate coverage-oriented inputs using symbolic execution or evolutionary algorithms. These strategies are more powerful, since a specific coverage target guides them. However, they do not take advantage of past exploration successes to dynamically learn the most compelling exploration strategy.

Reinforcement Learning (RL) is a machine learning approach that does not require a labeled training set as input since the learning process is guided by the positive or negative reward experienced during the tentative execution of the task. Hence, it represents a way to dynamically build an optimal exploration strategy by taking advantage of the past successful or unsuccessful moves.

RL has been extensively applied to the problem of GUI and Android testing [39, 42]. However, only the most basic RL (i.e., Tabular RL) has been applied to testing problems so far. In Tabular RL, the value of the state-action associations is stored in a fixed table. The advent of Deep Neural Networks (DNN) replaced Tabular approaches with Deep Learning ones, in which the action-value function is learned from the past positive and negative experiences made by one or more neural networks. When the state space to explore is extremely large (e.g., when an app has a significant amount of widgets), Deep RL has proved to be substantially superior to Tabular RL [9] [43] [32]. In this respect, we argue that the state space of Android apps is a good candidate for the successful adoption of Deep RL instead of Tabular RL for testing purposes.

This paper presents the first Deep RL approach, ARES, for automated *black-box* testing of Android apps. ARES uses a DNN to learn the best exploration strategy from previous attempts. Thanks to such DNN, it achieves high scalability, general applicability, and the capability to handle complex app behaviors.

ARES implements multiple Deep RL algorithms that come with a set of configurable, often critical, hyperparameters. To speed up selecting the most appropriate algorithm for the AUT and the fine-tuning of its hyperparameters, we have developed another tool, FATE, which integrates with ARES.

FATE is a simulation environment that supports a rapid assessment of Android testing algorithms by running *synthetic* Android apps (i.e., abstract navigational models of real Android apps). The execution of a testing session on a FATE synthetic app is, on average, 10 to 100 times faster than the execution of the same session on the corresponding real Android app.

We applied ARES to two benchmarks made by 41 and 68 Android apps, respectively. The first benchmark compares the performance of the ARES algorithms, while the latter evaluates ARES w.r.t. the state-of-the-art testing tools for Android.

Experimental results confirmed the hypothesis that Deep RL outperforms Tabular RL in exploring the state space of Android apps, as ARES exposed the highest number of faults and obtained

the highest code coverage. Furthermore, we carried out a qualitative analysis showing that the features of Android apps that make Deep RL particularly adequate include, among others, the presence of concatenated activities and blocking activities protected by authentication.

To sum up, this paper provides the following advancements to state of the art:

- ARES, the first publicly available testing approach based on Deep Reinforcement Learning, released as open-source;
- FATE, a simulation environment for fast experimentation of Android testing algorithms, also available as open-source;
- A thorough empirical evaluation of the proposed approach, whose replication package is publicly available to the research community.

2 Background

After a general overview on RL, this section presents in more detail Tabular RL and Deep RL.

2.1 Overview on Reinforcement Learning

The objective of Reinforcement Learning [47] is to train an *agent* that interacts with some environment to achieve a given goal. The agent is assumed to be capable of sensing the *current state* of the *environment*, and to receive a feedback signal, named *reward*, each time the agent takes an *action*.

At each time step t, the agent receives an observation x_t, takes an action a_t that causes the transition of the environment from state s_t to state s_{t+1}. A state s_t is a complete description of the state of the environment. An observation x_t is a partial representation of the state, which may omit information. The agent also receives a scalar reward $R(x_t, a_t, x_{t+1})$, that quantifies the goodness of the last transition.

For simplicity, let us assume $x_t = s_t$ [4]. The behavior of an agent is represented by a *policy* π, i.e., a rule for making the decision on what action to take, based on the perceived state s_t. A policy can be:

- Deterministic: $a_t = \pi(s_t)$, i.e. a direct mapping between states and actions;
- Stochastic: $\pi(a_t|s_t)$, a probability distribution over actions, given their state.

DDPG [33] and TD3 [16] are examples of RL algorithms that learn a deterministic policy, while SAC [25] is a RL algorithm that learns a stochastic policy.

The standard mathematical formalism used to describe the agent environment is a *Markov Decision Process (MDP)*. An MDP is a 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$, where :

- S is the set of all valid states,
- A is the set of all valid actions,
- $R : S \times A \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,
- $P : S \times A \to P(s)$ is the transition probability function, with $P(s_{t+1}|s_t, a_t)$ being the probability of transitioning into state s_{t+1} starting from state s_t and taking action a_t,
- $\rho_0(s)$ is the starting state distribution.

Markov Decision Processes obey the *Markov property*: a transition only depends on the most recent state and action (and not on states/actions that precede the most recent ones).

The goal in RL is to learn a policy π which maximizes the so-called *expected return*, which can be computed as:

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

A discount factor $\gamma \in (0, 1)$ is needed for convergence. When $t \to \infty$, if $\gamma = 1$, the expected return and the RL algorithm would not converge. The discount factor determines how much the agent cares about rewards in the distant future relative to those in the immediate future: if γ is small, the agent will tend to be myopic and only learn about actions that produce an immediate reward. If γ is close to 1, the agent will evaluate each of its actions based on its estimated future rewards. τ is a sequence of states and actions in the environment $\tau = (a_0, s_0, a_1, s_1...)$, named *trajectory* or *episode*. Testing an Android app can be seen as a task divided into finite-length episodes. Almost all reinforcement learning algorithms involve estimating *value functions*—functions of states (or *action-value functions*—functions of state-action pairs) [47] that estimate the goodness to perform a given action in a given state. The notion of goodness is defined in terms of expected future return. The rewards the agent can receive in the future depend on what actions it will take. The value function $V^\pi(s)$ is defined as the expected return starting in a state s and then acting according to a given policy π:

$$V^\pi(s) = E[G_t | s_0 = s]$$

The action-value function $Q^\pi(s, a)$ can be used to describe the expected return after taking an action a in state s and thereafter following the policy π:

$$Q^\pi(s, a) = E[G_t | s, a]$$

Correspondingly, we can define the *optimal value function*, $V^*(s)$, as the $V^\pi(s)$ that gives the highest expected return when starting in state s and acting according to the optimal policy in the environment. The *optimal action-value function*, $Q^*(s, a)$, gives the highest achievable expected return under the constraints that the process starts at state s, takes an action a and then acts according to the optimal policy in the environment.

A policy that chooses greedy actions only with respect to Q^* is optimal, i.e., knowledge of Q^* alone is sufficient for finding the optimal policy. As a result, if we have Q^*, we can directly obtain the optimal action, $a^*(s)$, via $a^*(s) = \text{argmax}_a Q^*(s, a)$. The optimal value function $V^*(s)$ and action-value function $Q^*(s, a)$ can be computed by means of a recursive relationship known as the *optimal Bellman equations*:

$$\begin{aligned} V^*(s_t) &= \max_a E[r(s_t, a_t) + \gamma V^*(s_{t+1})] \\ Q^*(s_t, a_t) &= E[r(s_t, a_t) + \gamma \max_{a_{t+1}} [Q^*(s_{t+1}, a_{t+1})]] \end{aligned}$$

The optimal Bellman equations constrains the value of a state under an optimal policy to be equal to the expected return for the best action from that state. The optimal Bellman equations are actually a system of equations, one for each state. If the MDP of the environment is known, then the system of equations can be solved analytically, finding the optimal value function (or the optimal action-value function) and, at last, the optimal policy. Otherwise approximate solutions are found iteratively.

2.2 Tabular RL

Tabular techniques refer to RL algorithms where the state and action spaces are approximated using value functions defined by means of tables. In particular, *Q-Learning* [49] is one of the most adopted algorithms of this family. Q-Learning aims to learn a so-called *Q-Table*, i.e., a table containing the value of each state-action pair. A Q-Table represents the current estimate of the action-value function $Q(s, a)$. Every time an action a_t is taken and a state s_t is reached, the associated state-action value in the Q-Table is updated as follows:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

where α is the learning rate (between 0 and 1). If the learning rate is close to 0 the agent exploits only prior knowledge, while when close to 1, the agent considers only the most up-to-date information. γ is the discount factor, applied to the future reward. Typically, γ ranges from 0.8 to 0.99 [41] [33] [16], while $\max_a Q(s_{t+1}, a)$ gives the maximum value for future rewards across all actions. It is used to update the reward for the current state-action pair.

RL algorithms based on the Tabular approach do not scale to high-dimensional problems, because in such cases it is difficult to manually define a good initial Q-Table. In case a good initial Q-Table is not available, convergence to the optimal table by means of the update rule described above is too slow [33].

2.3 Deep Reinforcement Learning

In large or unbounded discrete spaces, where representing all states and actions in a Q-Table is impractical, Tabular methods become highly unstable and incapable of learning a successful policy[41]. The rise of Deep Learning, relying on the powerful function approximation properties of Deep Neural Networks, has provided new tools to overcome these limitations. One of the first Deep Reinforcement Learning algorithms is DQN (Deep Q-Networks) [41].

DQN uses convolutional neural networks to approximate the computation of the action-value function Q^π. Training of such neural networks is achieved by using the so-called *experience replay* [34]. With experience replay the agent's experience e_t is stored within a buffer D named *replay buffer*. The experience e_t is defined as the tuple:

$$e_t = (s_t, a_t, r_t, s_{t+1}, d).$$

The tuple contains the state s_t of the environment, the action a_t taken in s_t, the reward r_t, and the next state of the environment s_{t+1}. The parameter d represents a binary value that indicates whether the transition has led the agent to a terminal state. To train the neural network, the algorithm occasionally retrieves random experience samples from the replay buffer. If the network learns only from consecutive samples of experience as they occurred sequentially in the environment, the samples would be highly correlated and lead to inefficient learning [41]. Instead, taking random samples from the memory replay breaks this correlation.

While DQN can indeed solve problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. The recent advancements over DQN described in the following paragraphs (namely, DDPG, TD3, and SAC) overcome such limitations and allow dealing with high-dimensional action spaces.

Deep Deterministic Policy Gradient (DDPG)

DDPG [33] is an *Actor-Critic* algorithm, i.e., it includes two roles: the *Critic*, which estimates the value function, and the *Actor*, which updates the policy π in the direction suggested by the Critic. It is based on a deterministic policy gradient [44] that can operate over continuous action spaces.

More specifically, the Critic has the objective of learning an approximation of the function $Q^*(s)$.

Suppose that the approximator is a neural network $Q_\phi(s, a)$, and that we have a set *D* of past experiences e_t $(s_t, a_t, r_t, s_{t+1}, d)$. The parameter list ϕ represents the coefficients of the neural network model, and the objective of the training phase is to optimize them, while set *D* is the previously seen *replay buffer*. The replay buffer should be large enough to avoid overfitting. The loss function of the neural approximator is the so-called *Mean-Squared Bellman Error (MSBE)*, which measures how close $Q_\theta(s, a)$ is to satisfying the Bellman equation:

$$L(\phi, D) = E[(Q_\phi(s_t, a_t) - (r_t + \gamma(1 - d) \max_{a_{t+1}} Q_\phi(s_{t+1}, a_{t+1})))^2]$$

The term subtracted from $Q_\phi(s_t, a_t)$ is named the *target*, because minimization of MSBE makes the Q-function as close as possible to this value. Since the target depends recursively on the same parameter ϕ to train, MSBE minimization can become unstable. The solution is to use a second neural network, called *target network*, whose parameters are updated to be close to ϕ, but with some time delay that gives stability to the process.

The Actor's goal is to learn a deterministic policy $\pi_\theta(s)$ which maximizes $Q_\phi(s, a)$. Because the action space is continuous and we assume the Q-function is differentiable with respect to the action, we can use gradient ascent with respect to the policy parameter. In case the action space is non-differentiable, DDPG degenerates, performing like DQN[33].

Twin Delayed DDPG (TD3)

Although DDPG can often achieve good performance, it tends to be susceptible to critical tuning of its hyperparameters. In fact, Fujimoto et al.[16] demonstrated that in Actor-Critic algorithms, such as DDPG, the policy update introduces errors, and overestimation of parameters ϕ. The researchers state that the introduced errors may be minimal, but raise two concerns: (1) the error may develop into a more significant bias over many updates if left unchecked, (2) an inaccurate value estimate may lead to poor policy update. TD3 [16] is an algorithm which addresses this issue by introducing three major changes, mostly on the Critic side: (1) *Clipped Double Q-Learning*: TD3 learns two Q-functions instead of one, and uses the smaller of the two Q-values as the target in the MSBE function. Using the smaller Q-value for the target, and regressing towards that, helps mitigate overestimation in the Q-function. (2) *Delayed Policy Update*: TD3 updates the policy and the target networks less frequently than the Q-function. Delaying policy updates reduces per-update error and further improves performance. (3) *Target Policy Smoothing*: TD3 adds noise to the target action to make it harder for the policy to exploit Q-function errors, by smoothing out Q across changes of the action.

Soft Actor Critic (SAC)

The central feature of SAC [25] is entropy regularization. Using the entropy-regularized method, an agent gets a bonus reward at each time step which is proportional to the entropy of the policy at that time step. In fact, differently from TD3, the policy of SAC is non deterministic and inclusion of entropy in the reward aims at promoting policies with a wider spread of alternatives to choose stochastically from. The RL problem becomes the problem of finding the optimal policy π^* according to the following equation:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, E[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi(\cdot|s_t)))].$$

The first term of the equation $E[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1})]$ comes from standard RL algorithms, in which the objective is to maximize the expected sum of rewards. The second term $\alpha H(\pi(\cdot|s_t))$ contains the entropy H, that is directly controlled by the entropy regularization coefficient $\alpha > 0$.

This parameter explicitly controls the explore-exploit trade-off. With higher α the exploration is encouraged, while lower α corresponds to more exploitation.

3 ARES: Approach

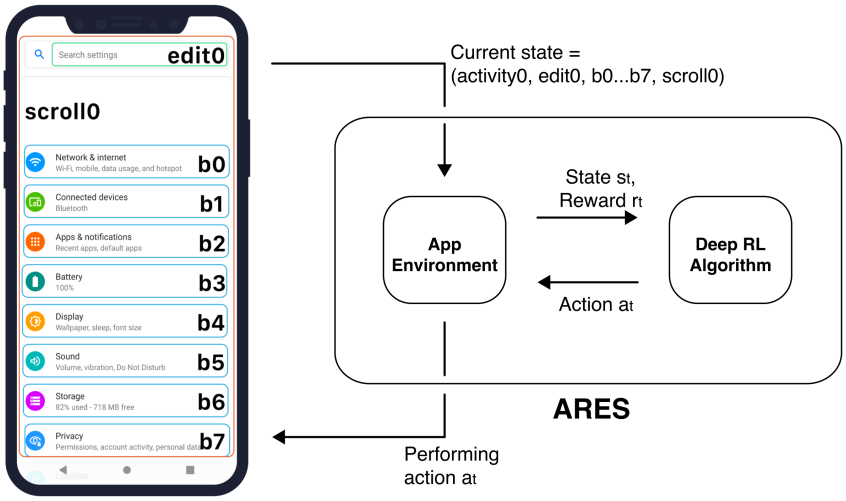

Figure 1: The ARES testing workflow. Information about widgets is extracted from the GUI and used to compute the state s_t and the reward r_t. b_i indicates the clickable buttons, $scroll_j$ indicates a portion of the GUI that we can scroll, and $edit_k$ an element that can be filled with text.

This section describes ARES (Application of REinforcement learning to android Software testing), our approach to black-box Android GUI testing based on Deep RL. Figure 1 shows an overview of the approach. The RL *environment* is represented by the Android application under test (AUT), which is subject to several interaction steps. At each time step, assuming the GUI state is s_t, and the reward is r_t, ARES first takes an action a_t. Then, it receives the new GUI state s_{t+1} of the AUT, and a reward r_{t+1} (not shown in Figure 1). Intuitively, if the new state s_{t+1} is similar to the prior state s_t, the reward is negative. Otherwise, the reward is a large positive value. In this way, ARES promotes the exploration of new states in the AUT, under the assumption that this is useful to test the application more thoroughly.

The reward is used to update the neural network, which learns how to guide the Deep RL algorithm to explore the AUT in depth. The actual update strategy depends on the Deep RL algorithm used (either DDPG, TD3, or SAC).

3.1 Problem Formulation

To apply RL, we have to map the problem of Android black-box testing to the standard mathematical formalization of RL: an MDP, defined by the 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$. Moreover, we have to map the testing problem onto an RL task divided into several finite-length episodes.

State Representation

Our approach is black-box because it does not access the source code of the AUT. It only relies on the GUI of the AUT. The state $s_t \in S$ is defined as a combined state $(a_0, ...a_n, w_0, ...w_m)$. The first part of the state $a_0, ...a_n$ is a one-hot encoding of the current activity, i.e., a_i is equal to 1 only if the currently displayed activity is the i-th activity, it is equal to 0 for all the other activities. In the second part of the state vector, w_j is equal to 1 if the j-th widget is available in the current activity; it is equal to 0 otherwise.

Action Representation

User interaction events in the AUT are mapped to the action set A of the MDP. ARES infers executable events in the current state by analyzing the dumped widgets and their attributes (i.e., *clickable*, *long-clickable*, and *scrollable*). In addition to the widget-level actions we also use two system-level actions, namely *toggle internet connection* and *rotate screen*. These system-level actions are the only system actions that can be easily tested. In fact, since Android version 7.0, testing other system-level actions (like those implemented in Stoat [45]) would depend on the Android version used [46], [21], and would require a rooted device [18].

Moreover, certain apps, such as apps in the Finance and Entertainment categories (e.g., Netflix, SkyGo, Amazon Prime, and Lloyds), do not start on a rooted device due to the root checks that are on the apps [48] [19] [40], compromising their testing.

Each action a is 3-dimensional: the first dimension represents the widget ARES is going to interact with or the identifier of a system action. The second dimension specifies a string to be used as text input if needed. Actually, an index pointing to an entry in a dictionary of predefined strings is used for this dimension. The third dimension depends on the context: when the selected widget is both *clickable* and *long-clickable*, the third action determines which of the two actions to take. When ARES interacts with a *scrollable* object, the third dimension determines the scrolling direction.

Transition Probability Function

The transition function P determines which state the application can transit to after ARES has taken an action. In our case, this is decided solely by the execution of the AUT: ARES observes the process passively, collecting the new state after the transition has taken place.

Reward Function

The RL algorithm used by ARES receives a reward $r_t \in R$ every time it executes an action a_t. We define the following reward function:

$$r_t = \begin{cases} \Gamma_1 & \text{if } act(s_{t+1}) \notin act(E_t) \text{ or } \textit{crash} \\ -\Gamma_2 & \text{if } pack(act(s_{t+1})) \neq pack(AUT) \\ -\Gamma_3 & \text{otherwise.} \end{cases} \tag{1}$$

with $\Gamma_1 \gg \Gamma_2 \gg \Gamma_3$ (in our implementation $\Gamma_1 = 1000$, $\Gamma_2 = 100$, $\Gamma_3 = 1$). To select the values of the reward, we conducted a preliminary experiment on a sub-sample of five apps (i.e., Antennapod, RedReader, Opentasks, Simple-Solitaire and YalpStore). On these apps, we tested several types of rewards that are commonly used in literature: combination I ($\Gamma_1 = 1000$, $\Gamma_2 = 100$, $\Gamma_3 = 1$), combination II ($\Gamma_1 = 1$, $\Gamma_2 = 1$, $\Gamma_3 = 0$), and combination III ($\Gamma_1 = 100$, $\Gamma_2 = 10$, $\Gamma_3 = 1$). We found that combination I gives the best results.

The exploration of ARES is divided into *episodes*. At time t, the reward r_t is high (Γ_1) if ARES was able to transition to an activity never observed during the current episode E_t (i.e., the next activity $act(s_{t+1})$ does not belong to the set of activities encountered so far in E_t): if a new episode is started at $t + 1$, its set of activities is reset: $act(E_{t+1}) = \emptyset$.

Resetting the set of encountered activities at the beginning of each new episode is a technique that encourages ARES to visit and explore the highest number of activities in each episode to reinforce its explorative behaviors continuously. In contrast, if we provide the algorithm a significant reward only a few times (i.e., "sparsely"), the information to learn the optimal state-action combinations might be insufficient. The algorithm might fail to reproduce the sequence of actions leading to a high reward in the future. In that case, the performance of the algorithm results poor. On

the contrary, rewarding any activity change, regardless of their novelty, would encourage cycling behaviors [13].

The reward is high (Γ_1) also when a faulty behavior (*crash*) occurs. It is very low ($-\Gamma_2$) when the displayed activity does not belong to the AUT (i.e., the package of the current activity, $pack(act(s_{t+1}))$, is not the package of the AUT), as we aim to explore the current AUT only. In all other cases, the reward is moderately negative ($-\Gamma_3$), as the exploration remains inside the AUT, even if nothing new has been discovered.

4 ARES: Implementation

ARES features a custom environment based on the OpenAI Gym [10] interface, which is a de-facto standard in the RL field. OpenAI Gym is a toolkit for designing and comparing RL algorithms and includes a number of built-in environments. It also includes guidelines for the definition of custom environments. Our custom environment interacts with the Android emulator [20] using the Appium Test Automator Framework [3].

4.1 Tool Overview

As soon as it is launched, ARES leverages Appium to dump the widget hierarchy of the GUI in the starting activity of the AUT. The widget hierarchy is analyzed by searching for clickable, long-clickable, and scrollable widgets. Afterward, these widgets are stored in a dictionary containing several associated attributes (e.g., resource-id, clickable, long-clickable, scrollable, etc.) and compose the *action vector*, i.e., the vector of executable actions in the current state. At each time step, ARES takes an action according to the behavior of the exploration algorithm. Once the action has been fully processed, ARES extracts the new widget hierarchy from the current GUI and calculates its MD5 hash value. If it has the same MD5 value of the previous state, ARES leaves the action vector unchanged. If the MD5 value does not match, ARES updates the action vector. ARES performs the observation of the AUT state and returns the combined vector of activities and widgets. ARES organizes the testing of each app as a task divided into finite-length episodes. The goal of ARES is to maximize the total reward received during each episode. Every episode lasts at least 250 time steps. Its duration is shorter only if the app crashes. To select the ideal episode boundaries, we conducted a preliminary experiment on a sample app. On this app, we trained the same algorithm by varying the episode length. Training characterized by short episodes results in poor performance due to the impossibility of exploring the application. Similarly, long episodes suffered from poor performance due to a low number of episodes completed. Once an episode comes to an end, the app is restarted, and ARES uses the acquired knowledge to explore the app more thoroughly in the next episode.

4.2 Application Environment

The application environment is responsible for handling the actions to interact with the AUT. Since the environment follows the guidelines of the Gym interface, it is structured as a class with two key functions. The first function `init(desired_capabilities)` is the initialization of the class. The additional parameter `desired_capabilities` consists of a dictionary containing the emulator setup and the application to be tested. The second function is the `step(a)` function, that takes an action `a` as command and returns a list of objects, including `observation` (current AUT state) and `reward`.

4.3 Algorithm Implementation

ARES is a modular framework that adopts a plugin architecture to integrate the RL algorithm to use. Hence, extension with a new exploration algorithm can be easily achieved. In the current implementation, ARES provides five different exploration strategies: (1) random, (2) Q-Learning, (3) DDPG, (4) SAC, (5) TD3. The random algorithm interacts with the AUT by randomly selecting an action from those in the action vector. Compared to Monkey [22], our random approach performs better since it selects only actions from the action vector. In fact, Monkey generates random, low-level events on the whole GUI, which could target no actual widget and then be discarded.

Our Q-Learning strategy implements the algorithm proposed by Watkins and Dayan [49]. The Deep RL algorithms available in ARES are DDPG, SAC, and TD3. Their implementation comes from the Python library *Stable Baselines* [26], and allows ARES to save the status of the neural network as a policy file at the end of the exploration. In this way, the policy can be loaded and reused on a new version of the AUT at a later stage, rather than restarting ARES from scratch each time a new AUT version is released. ARES is publicly available as open source software at `https://github.com/H2SO4T/ARES`.

4.4 Compatibility

ARES has been successfully tested on Windows 10, macOS 11.1 (and older), Ubuntu 20 (and older), and Scientific Linux 7.5. ARES is fully compliant with parallel execution and enables parallel experiments to be performed on emulators or real devices, handling each instance in a completely separate manner. ARES is also compatible with several Android versions (i.e., it has been successfully tested on Android 6.0, 7.0, 7.1, 8.0, 8.1, 9.0, and 10.0). Moreover, since ARES is based on the standard OpenAIGym, new algorithms and exploration strategies can be easily added to the tool.

5 Fast Android Test Environment (FATE)

Deep RL algorithms require fine-tuning, which is expensive on real apps. Therefore, we developed FATE, a simulation environment for fast Android testing. FATE models only the navigation constraints of Android apps, so it can efficiently compare alternative testing algorithms and quickly tune their corresponding hyperparameters. After this algorithm selection and tuning phase through FATE is completed, the selected algorithms and their configurations are ported to ARES to test real apps.

5.1 Model-Based Prototyping

5.1.1 FATE Design

In FATE, developers model an Android app by means of a deterministic Finite State Machine (FSM) $\mathcal{F} = (\Sigma, S, s_0, \delta, F)$, where Σ is a set of events, S a set of states with s_0 the initial state and F the set of final states, and δ the state transition function $\delta : S \times \Sigma \longrightarrow 2^S$. The *states* S of the FSM correspond to the activities of the app, while the *events* Σ trigger the transitions between activities, which in turn are modeled as a transition table δ. Events represent the clickable widgets (e.g., buttons) available in each activity. Transitions have access to a set of *global variables* and possess, among others, the following attributes: *ID*, *type*, *active* (boolean attribute), *guard* (boolean expression that prevents the transition from being taken if it evaluates to false), *set* (new values to be assigned to global variables), *destination* (target activity, i.e., value of δ). A prototype of a FATE model is shown in figure 2.

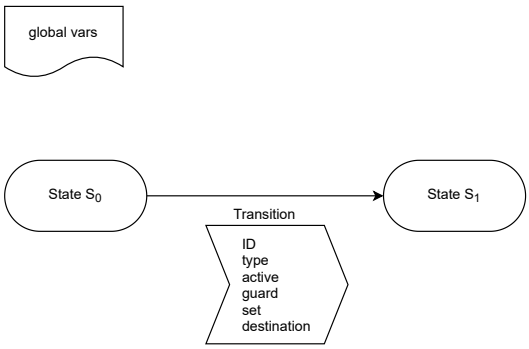

Figure 2: Prototype of a FATE model.

5.1.2 FATE Implementation

Figure 3 shows the FATE model of the prototypical app *Social Network*. To build such a model, developers can use Ptolemy [30] to graphically draw an FSM that mimics the behavior of the application. While creating an FSM with Ptolemy is not mandatory in FATE, it simplifies the job of designing a logically correct model, thanks to the checks it performs. Then, FATE automatically translates the Ptolemy model (saved in XML format) into a JSON file that replicates the structure and behavior of the Ptolemy model. The JSON model translation has two main fields: `global_vars` and `nodes`. The first contains a list of global variables organized by `name` and `value`. The latter contains a list of all the activities. Each activity is characterized by a `node_id` and a list of corresponding node transitions, each including all the respective transition attributes.

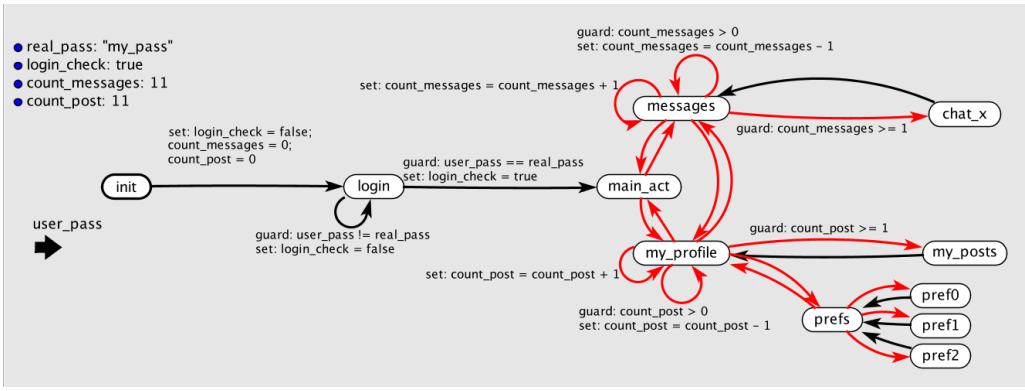

Figure 3: FATE model of *Social Network*: global variables are shown on the top-left; inputs on the bottom-left; red edges indicate non deterministic transitions.

The model of *Social Network* in Figure 3 contains a transition from `login` to `main_act` which is subjected to the guard `user_pass == real_pass`, i.e., the entered password must be correct in order for the transition to be taken. In the JSON model, such transition is coded as:

```
"transition": {
    "transition_id": 0,
    "type": "button",
    "active": true,
    "guard": "user_pass == real_pass",
    "set": null,
    "destination": "main_act"
}
```

Another example of guarded transition is the one between `messages` and `chat_x`. The guard `count_messages >= 1` checks whether there exists at least one message from which a chat thread can be started. In the JSON model this is coded as:

```
"transition": {
    "transition_id": 0,
    "type": "button",
    "active": true,
    "guard": "count_messages >= 1",
    "set": null,
    "destination": "main_act"
}
```

In FATE, a Python environment compliant with the OpenAI Gym standard takes as input the JSON app model and tests it automatically using the selected algorithm. The available algorithms are: (1) Random, (2) Q-Learning, (3) DDPG, (4) SAC, (5) TD3. FATE was built with modularity in mind, and new exploration algorithms can be easily added to the tool. Compared to testing an Android app through Espresso or Appium, FATE makes test case execution significantly faster because there is no need to interact with the app via its GUI. Moreover, the application navigation logic is simulated by the transition function δ, making it usually much faster to execute. Consequently, developers can run a large number of experiments, evaluate multiple algorithms, check various algorithm or application configurations, and find the optimal set of hyperparameters, all of which would be prohibitively expensive to execute on a standard Android testing platform. A limitation of FATE is that its effectiveness in hyperparameter tuning depends on the fidelity of the app models created by the developers. Despite not being the developers of the apps under test, we have been able to define models for them that turned out to be sufficiently faithful. In fact (see Section 6.1), the results obtained on the real apps with ARES and on their models with FATE are very close to each other. FATE is publicly available as open source software at `https://github.com/H2SO4T/ARES`.

5.2 Representative Family of Models

For fast evaluation of the Deep RL algorithms implemented in ARES, we modeled four Android apps using FATE. Each model represents the generalization of the apps belonging to a specific family, such as Shopping category. To obtain a set of app models representing the most common apps used in everyday life, we inspected AppBrain (a website that aggregates Google Play Store statistics and rankings) [2]. We selected four different and representative categories from the top ten: Music & Audio, Lifestyle, Business, and Shopping. From each category we then selected and modeled in FATE one prototypical app: *Player*, *Social Network*, *Bank* and *Market Place*. Each model is configurable with a variable degree of complexity.

The simplest scenario is *Player*. It features a wide number of activities arranged in a tree-like structure. It reflects the generalization of various applications, including apps or app components, to manage the settings and stream/add/remove media contents. *Social Network* (see Figure 3) starts by prompting a login activity with fields for username and password. Following the login activity, we have several activities that replicate a standard social network behavior, including a basic chat activity. The presence of inner password-protected operations characterizes the *Bank* model. *Market Place* models a typical app for e-commerce: the user can search for goods, login, purchase products, and monitor the orders. The four representative app models used in this work are publicly available inside the FATE tool.

6 Evaluation

We seek to address the following research questions, split between the following two studies:
Study 1 (FATE):

- **RQ1** *Are the results of synthetic apps comparable to those of the their translated counterparts?*

- **RQ2** *Which Deep RL algorithm and which algorithm configuration performs better on the synthetic apps?*
- **RQ3** *How does activity coverage vary as the model of the AUT becomes increasingly difficult to explore?*
- **RQ4** *What are the features of the synthetic apps that allow Deep RL to perform better than Q-Learning?*

In Study 1, we want to understand if results obtained on synthetic app models run by FATE correlate with those obtained when executing the same apps (RQ1), once they are translated into real Android apps in Java code that can be executed by ARES. In particular, we translated three synthetic apps to Java/Android: Social Network, Bank, and Market Place. We decided not to translate the "Player" model due to its simplicity. We compare the rankings of the algorithms produced respectively by FATE and by ARES. Text inputs are chosen from a dictionary of 40 strings, which include credentials necessary to pass through the login activities. Since in this study we also use synthetic apps generated from models (i.e., Player, Social Network, Bank, and Market Place), coverage is measured at the granularity of Android activities. In fact, there is no source code implementing the business logic. Each run has a length of 4000 time steps, close to an hour of testing in a real Android test setting. With FATE, 4000 times steps are executed approximately in 200 seconds.

To answer RQ2, we take advantage of the fast execution granted by FATE to compare alternative RL algorithms on all synthetic apps and determine their optimal configuration (see Appendix 1). Text inputs are chosen from a dictionary of 20 strings, which include credentials necessary to pass through the login activities. To account for non-determinism, we executed each algorithm 60 times for each hyperparameter configuration of the algorithms and applied the Wilcoxon non-parametric statistical test to draw conclusions on the difference between algorithms and configurations, adopting the conventional p-value threshold at $\alpha = 0.05$. Since multiple pairwise comparisons are performed with overlapping data, the chance to reject true null hypotheses may increase (type I error). To control this problem, we adopt the Holm-Bonferroni correction [27], which consists of using more strict significance levels when the number of pairwise tests increases.

To answer RQ3, we consider the best performing configuration of the Deep RL algorithms, as selected from RQ2, and gradually increase the exploration complexity of the apps. Specifically, *20_strings, 40_strings, 80_strings* indicate an increasing size of the *string pool*. Such *string pool* is a dictionary of 20, 40, or 80 strings containing numbers and words, including the app's username and password, to use with a login activity. The string pool does not contain duplicates. *augmented_5* and *augmented_10* indicate an increasing size of the self navigation links (with 5 or 10 "dummy" buttons that do nothing) within the login activities.

For the assessment, we adopt the widely used metric AUC (Area Under the Curve), measuring the area below the activity coverage plot over time. To account for the non-determinism of the algorithms, we repeated each experiment 30 times and applied the Wilcoxon non-parametric statistical test. In RQ4, we investigate qualitatively the cases where Deep RL is superior to Tabular Q-Learning.

Study 2 (ARES):

- **RQ5** *How do code coverage and time-dependent code coverage compare between Random, Q-Learning, DDPG, and SAC?*
- **RQ6** *What are the fault exposure capabilities of the alternative approaches?*
- **RQ7** *What features of the real apps make Deep RL perform better than Q-Learning?*
- **RQ8** *How does ARES compare with state-of-the-art tools in terms of coverage and bug detection?*

In Study 2, we use real apps and compare the alternative Deep RL algorithms between each other and with Random and Tabular Q-Learning. At last, we compare ARES to state-of-the-art testing tools.

To address RQ5-RQ6 and RQ7, we randomly selected 100 apps among the 500 most starred F-Droid apps available on GitHub, and 41 successfully compiled. We consider coverage at the source code level and compare both the final coverage and the coverage increase over time (RQ5). To obtain coverage data at the instruction level, we instrumented each app using JaCoCo [38]. As in Study 1, we measured AUC with respect to the code coverage and compared AUC values using the Wilcoxon statistical test with a significance level set to 0.05 (with correction). We exclude TD3 from the comparison, since it performed consistently worse than the other RL algorithms on synthetic apps.

In addition to code coverage, we also report the number of failures (unique app crashes) triggered by each approach (RQ6). To measure the number of unique crashes observed, we parsed the output of Logcat and (1) removed all crashes that do not contain the package name of the app; (2) extracted the stack trace; (3) computed the hash code of the sanitized stack trace, to uniquely identify it. With RQ7, we analyze the different performances of Deep RL vs. Q-Learning on real apps qualitatively.

To address RQ8, we evaluate and compare ARES and state-of-the-art tools in terms of code coverage and the number of crashes, using two different sets of apps under test, RQ8-a and RQ8-b, that accommodate the different requirements and constraints of the tools being compared. As state-of-the-art tools, we selected Monkey [22], Sapienz [37], TimeMachine [15] and Q-Testing [42]. In RQ8-a, we compare ARES, Monkey, Sapienz, and TimeMachine on a set of 68 apps coming from AndroTest [11]. These apps are instrumented using Emma [1], the same coverage tool that is used in Sapienz and TimeMachine. In RQ8-b, we compare ARES to Q-Testing on a set of ten apps instrumented using JaCoCo, the coverage tool supported by Q-Testing.

All experimental data were generated and processed automatically. Each experiment was conducted with a one-hour timeout and was repeated ten times for a total of 4560 hours ($\approx$ 190 days). The emulators involved in the study are equipped with 2 GB of RAM and Android 10.0 (API Level 29) or Android 4.4 only for the tool comparison.

6.1 Experimental Results: Study 1

Table 1 shows the ranking of the algorithms produced by ARES vs. FATE on the three apps translated from the synthetic FATE models to Java/Android. Below the ranking, Table 1 shows the AUC values obtained by the respective algorithms. The behaviors of the considered algorithms on synthetic (FATE) vs. translated (ARES) apps are very similar. The AUC values are quite close, and Spearman's correlation between AUC values across algorithms is 0.99 for Social, 0.89 for Bank, and 0.99 for Market; it is 0.95 overall. All correlations are statistically significant at level 0.05. ARES required 450 hours to complete the experiments. FATE required around 10 hours, reducing the computation time by a factor of 45 while producing similar results as ARES.

> RQ1: The results obtained on synthetic apps are comparable to those obtained on their translated counterparts.

Figure 4 shows the coverage growth for the synthetic app Social. Each curve shows the mean of 60 runs. The shaded area around the mean represents the Standard Error of the Mean (*SEM* = $\sigma/\sqrt{n}$, where σ is the standard deviation and $n = 60$ the number of points). The highest activity coverage is obtained consistently by Deep RL algorithms, which have higher AUC values. Table 2 reports the AUC obtained on the synthetic apps in all tested configurations. Table 2 also shows the Vargha-Delaney effect size in the case of a statistically significant p-value $< \alpha/k$ where k is computed from the Holm-Bonferroni correction for multiple tests, between the winner algorithm

App	Tool	Ranking / AUC				
Social	ARES	Q-Learn	DDPG	Rand	SAC	TD3
		4: 7788	5: 6802	3: 9547	2: 9594	1: 15101
	FATE	Q-Learn	DDPG	Rand	SAC	TD3
		4:7737	5:7363	3:9291	2:10361	1:14451
Bank	ARES	Q-Learn	DDPG	Rand	SAC	TD3
		4: 8614	5: 7976	3: 9344	1: 12138	2: 10932
	FATE	Q-Learn	DDPG	Rand	SAC	TD3
		4: 7750	5: 6458	2: 9746	1: 16535	3: 9305
Market	ARES	Q-Learn	DDPG	Rand	SAC	TD3
		1: 16866	2: 16788	4: 15936	5: 15930	3: 15944
	FATE	Q-Learn	DDPG	Rand	SAC	TD3
		1: 16496	2: 16318	4: 15943	5: 15936	3: 15949

Table 1: Ranking of algorithms produced by ARES vs FATE; AUC values below ranked algorithms.

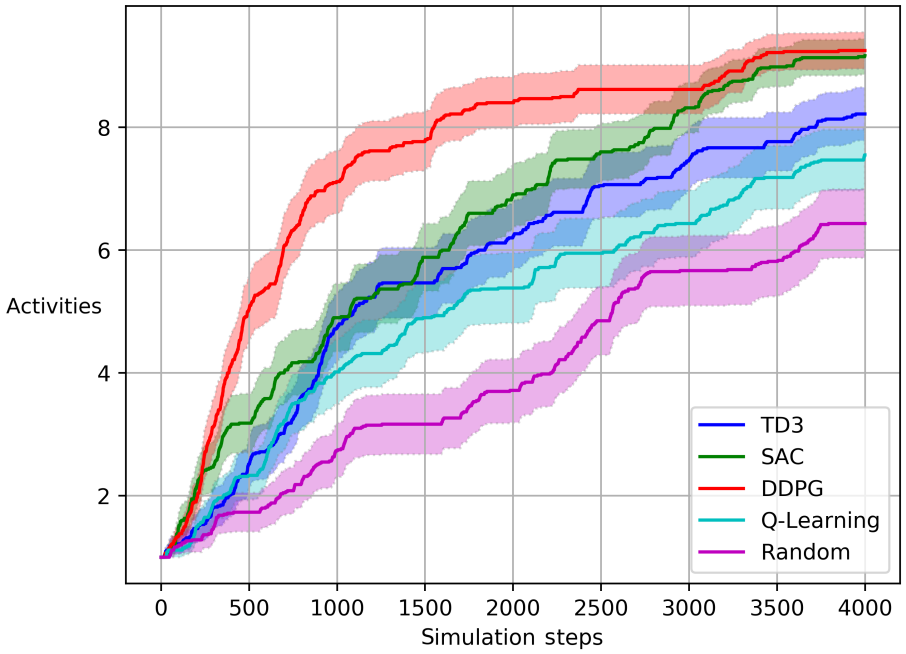

Figure 4: Activity coverage of the Social synthetic app in FATE.

(highest AUC) and the remainders.

Results show that Deep RL algorithms achieve higher coverage in most experiments. DDPG performs better in the simplest configuration, *20_strings*, while SAC performs better in almost all other configurations, including the most complex ones. Q-Learning prevails in only two scenarios belonging to *Market Place*, but the difference from the other algorithms is not statistically significant (p-value $> \alpha$).

RQ2: Results lead us to select DDPG and SAC as best performing Deep RL algorithms to be involved in Study 2 experiments.

DDPG is selected due to its high performance in relatively simple scenarios; SAC because of its ability to adapt and maintain good performance in the majority of scenarios.

App	Config	Rand	Q-Learn	TD3	SAC	DDPG	Effect Size
Player	*20_str*	88719	89022	89903	89943	90337	-
Social	*20_str*	15840	20387	22809	25463	30008	L(Rand), M(Q)
	40_str	9291	7737	14451	10361	7363	S(DDPG)
	80_str	4535	5640	5730	7254	4774	-
	aug_5	13960	15400	13094	17402	13385	-
	aug_10	5291	3998	13737	11559	8870	M(Q, Rand)
Bank	*20_str*	22894	21622	29159	28016	36977	M(Q, Rand)
	40_str	9746	7750	9305	16535	6458	S(Q, DDPG)
	80_str	3998	4843	4776	5621	4798	-
	aug_5	12815	8634	8702	14914	11472	-
	aug_10	4121	6289	13289	14195	15361	M(Q, Rand)
Market	*20_str*	19236	18471	20980	23403	25923	-
	40_str	15943	16496	15949	15936	16318	-
	80_str	15944	15945	15935	15937	15932	-
	aug_5	18917	16377	16500	21208	16027	-
	aug_10	4121	6289	13289	14195	15361	-

Table 2: Mean AUC for synthetic apps: effect size between the winner (shaded cell) and other algorithms is reported only when p-value is statistically significant (S = Small; M = Medium; L = Large).

RQ3: While in simple situations (e.g., the Player app), all algorithms achieve a high level of coverage, when things get more complex (e.g., when the string pool increases), Deep RL algorithms retain higher coverage than all other algorithms.

We have manually inspected the step-by-step exploration performed by Q-Learning and by the Deep RL algorithms. We found that login activities complicate substantially the exploration performed by Q-Learning. In fact, it is more difficult to reproduce the right username-password combination for a Tabular Q-Learning algorithm, which has limited adaptation capabilities. In contrast, Deep RL algorithms memorize the right combination in the DNN used to guide the exploration. In addition, large action spaces make it challenging for Q-Learning to learn an effective exploration strategy. The DNNs used by Deep RL algorithms can easily cope with large spaces of alternatives to choose from. The performance degradation of Q-Learning confirms this as the string pool increases in dimension or as new interactive elements ("dummy" buttons) are added, which confuse Q-Learning during its exploration.

RQ4: The performance of Q-Learning declines in the presence of blocking activities that require specific input combinations that must be learned from past interactions or when the input/action space becomes excessively large, while Deep RL can learn how to cope with such obstacles thanks to the DNN employed to learn the exploration strategy.

6.2 Experimental Results: Study 2

Table 3 shows coverage and crashes produced by each algorithm deployed in ARES. The highest average coverage and average number of crashes over ten runs are shaded in gray for each app. We grouped the apps into three different size categories (Low-Medium, Medium, and High), depending on their ELOC (Executable Lines Of Code). Results show that the Deep RL algorithms arise more often as winners when the ELOC increase. Usually, larger size apps are more sophisticated and offer a richer set of user interactions, making their exploration more challenging for automated tools. We already know from Study 1 that when the action space or the observation space of the apps increase, Deep RL can infer the complex actions needed to explore such apps more easily than other algorithms. Study 2 confirms the same trend.

Overall, SAC achieves the best performance, with 42.61% instruction coverage and 0.3 faults detected on average. DDPG comes next, with 40.09% instruction coverage and 0.12 faults detected

Applications	ELOC	Rand	Q	SAC	DDPG	Rand	Q	SAC	DDPG
		%Coverage(mean)				#Crashes(mean)			
Silent-ping-sms	263	41	41	41	41	0	0	0	0
Drawablenotepad	452	20	21	26	25	0.7	0.6	0.7	0.1
SmsMatrix	466	23	20	24	22	1.2	0	1.2	0.9
Busybox	540	75	73	74	76	0	0	0	0
WiFiKeyShare	627	37	36	37	37	0	0	0	0
Talalarmo	1094	69	71	71	71	0	0.5	0.5	0
AquaDroid	1157	55	55	55	55	1.0	0.4	0.8	0.3
Lexica	1215	72	72	74	75	0.3	0.1	1.5	1.2
Loyalty-card-locker	1228	41	37	50	41	0.5	0.4	0.8	0.1
Dns66	1264	58	58	58	58	0.1	0	0	0.2
Gpstest	1311	47	46	47	46	0	0	0	0
Memento	1336	77	76	74	77	0	0	0	0
Editor	1547	50	46	51	50	0	0	0	0
AndOTP	1560	20	25	27	20	0.5	0.5	0.7	0.2
BookyMcBookface	1595	26	25	25	24	0	0	0	0
Tuner	2207	80	74	79	75	0	0	0	0
WifiAnalyzer	2511	78	75	80	79	0	0	0	0
AdAway	3064	38	37	45	40	0	0	0.1	0.1
Gpslogger	3201	36	31	32	28	0	0	0	0.1
Connectbot	3904	26	25	28	18	0	0	0	0
Neurolab	3954	29	28	29	28	0	0.4	0.3	0.6
Anuto	4325	46	46	47	47	0	0	0	0
PassAndroid	4569	1	1	1	1	0	0	0	0
Markor	4607	51	43	53	41	0.3	0	0.4	0
Vanilla	4747	29	34	41	33	0	0	0	0
Average		45	43.84	46.76	44.32	0.15	0.12	0.28	0.15
Afwall	5130	12	12	16	13	0	0	0	0
OpenTracks	5260	45	42	44	45	0	0	0	0
Opentasks	5772	43	50	53	44	0	0	0.2	0
UserLAnd	5901	60	60	60	60	0.1	0.2	0.4	0.2
Simple-Solitaire	5907	10	30	31	31	0	0.4	0.4	0.2
Authorizer	5923	5	5	5	5	0	0	0	0
YalpStore	6734	35	34	38	33	0	0	0	0
CameraRoll	6836	32	31	31	32	0.8	0.1	1.6	0.1
AntennaPod	7975	46	40	48	38	0.5	0.1	0.8	0.4
Phonograph	8758	16	16	16	16	0	0	0	0
Average		30.4	30.5	34.2	31.7	0.14	0.08	0.34	0.09
MicroMathematics	10506	35	35	47	41	0	0	0	0
LightningBrowser	11961	35	36	43	37	0	0	0.4	0.1
Firefox-focus	12482	33	34	41	35	0.5	0.3	0.8	0.1
RedReader	12958	42	42	44	46	0	0	0.1	0
Wikipedia	23543	42	43	44	41	0	0	0	0
Slide	30483	19	17	18	19	0.8	0.3	1.2	0.3
Average		34.33	34.50	39.5	36.5	0.21	0.1	0.38	0.1
Total Average		39.62	39.58	42.61	40.09	0.17	0.1	0.3	0.12
Unique crashes						73	43	102	52

Table 3: Average coverage and number of crashes observed on 41 real open-source apps in 10 runs of ARES.

on average. To further investigate these results, we computed the AUCs reached by each algorithm, and we applied the Wilcoxon test to each pair of algorithms. Table 4 shows the AUCs achieved by the four algorithms and the Vargha-Delaney effect size between the winner and the other algorithms when the p-value is less than α. SAC results as the winner on 56% of the considered real apps, followed by Random (34%). Moreover, Table 4 confirms the trend observed in Table 3: as ELOC increase, a higher proportion of Deep RL algorithms produces the highest AUC. Figure 5 shows an example of code coverage over time for the app *Loyalty-card-locker*, averaged on 10 runs. SAC increases its coverage almost until the end of the exploration, while the other algorithms reach a plateau after around 35 minutes of exploration.

RQ5: SAC reached the highest coverage in 24/41 apps, followed by DDPG (11 apps), Random (10 apps), and Q-Learning(1 app). SAC also has the highest AUC in 24/41 apps, followed by Random (13 apps), Q-Learning (11 apps), and DDPG (5 apps).

We suspect that the higher performance of SAC is related to its entropy regularization parameter. Thanks to the entropy regularization, in contrast to the other Deep RL algorithms that do not contain such parameter, SAC can maintain a high level of exploration even at the end of the testing phase, preventing the policy from converging to a bad local optimum.

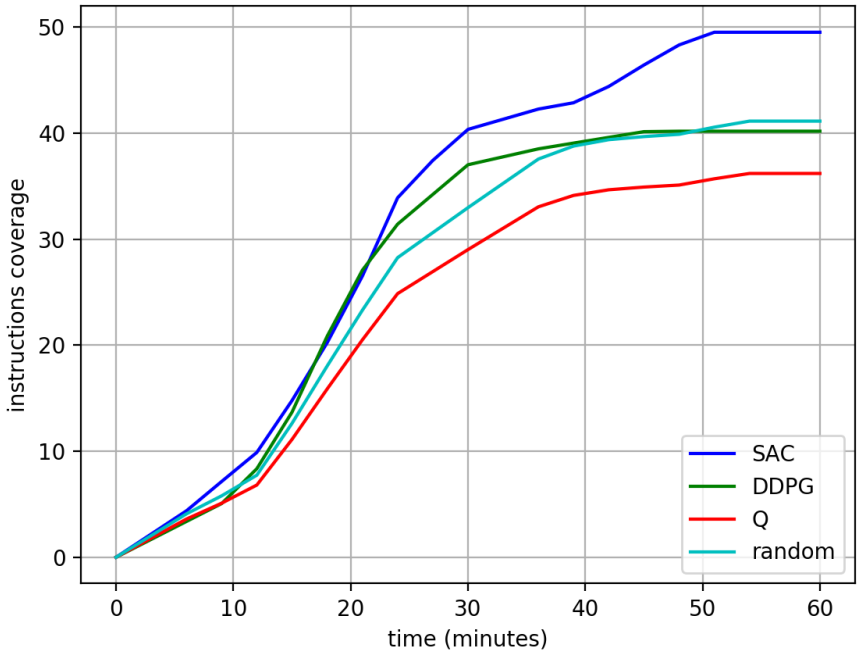

Figure 5: Instruction coverage over time for the app *Loyalty-card-locker*.

Table 3 shows that SAC exposed the highest number of unique crashes (102), followed by Random (73), DDPG (52) and Q-Learning (43). The most common types of error exposed by SAC during testing are: *RuntimeException* (34 occurrences), *NullPointerException* (14), *IllegalArgumentException* (13). Figure 6 shows around a thirty percent overlap between the crashes found by SAC and the other algorithms. The overlap with Random is the highest. SAC discovers about 40% of unique crashes found by Random; however, SAC found many new crashes that Random did not find.

RQ6: The SAC algorithm implemented in ARES generates the highest number of crashes, 102, in line with the results on coverage (RQ5), where SAC was also the best performing algorithm.

We have manually inspected the coverage progress of the different algorithms on some of the real apps considered in Study 2. We have identified two structural patterns: concatenated activities (i.e., a sequence of nested activities possibly requiring some precondition to move from one to the next) and blocking activities (activities that require a specific input combination to enable the transition to the next activity). We observed that Deep RL algorithms achieve higher coverage than the other algorithms when it is necessary to replicate complex behaviors in order to: (1) overcome blocking activities, e.g., to create an item in order to be able to access its properties later, or to successfully authenticate within the app; (2) to pass through concatenated activities without being

App	Rand	Q	SAC	DDPG	Effect Size
Silent-ping-sms	0.36	0.36	0.38	0.37	S(DDPG),M(Rand),L(Q)
Drawable-notepad	0.13	0.14	0.20	0.18	L(Rand,Q)
SmsMatrix	0.13	0.13	0.11	0.11	-
Busybox	0.54	0.56	0.68	0.68	L(Q,Rand)
WiFiKeyShare	0.29	0.29	0.33	0.30	L(DDPG,Q,Rand)
Talalarmo	0.62	0.60	0.64	0.64	L(Q)
AquaDroid	0.529	0.526	0.531	0.522	L(Rand, Q, DDPG)
Lexica	0.63	0.61	0.66	0.65	L(Q,Rand)
Loyalty-card-locker	0.23	0.23	0.34	0.21	L(DDPG,Q,Rand)
Dns66	0.51	0.51	0.47	0.45	L(DDPG,SAC)
Gpstest	0.40	0.40	0.39	0.36	L(DDPG)
Memento	0.64	0.65	0.64	0.65	-
Editor	0.42	0.37	0.37	0.37	L(DDPG,Q,SAC)
AndOTP	0.16	0.18	0.23	0.15	M(Rand), L(DDPG)
BookyMcBookface	0.22	0.20	0.20	0.20	M(DDPG), L(SAC)
Tuner	0.68	0.66	0.60	0.60	L(DDPG,SAC)
WifiAnalyzer	0.56	0.56	0.67	0.58	L(DDPG,Q,Rand)
AdAway	0.25	0.25	0.27	0.25	-
Gpslogger	0.28	0.28	0.23	0.20	L(DDPG,SAC)
Connectbot	0.19	0.19	0.22	0.09	L(DDPG,Q,Rand)
Neurolab	0.23	0.23	0.22	0.22	-
Anuto	0.35	0.40	0.43	0.33	L(DDPG,Q,Rand)
PassAndroid	0.018	0.017	0.017	0.017	-
Markor	0.40	0.40	0.43	0.25	L(DDPG,Q,Rand)
Vanilla	0.17	0.23	0.26	0.23	L(Rand)
Afwall	0.09	0.09	0.13	0.10	L(DDPG,Q,Rand)
OpenTracks	0.37	0.35	0.35	0.35	-
Opentasks	0.18	0.46	0.46	0.29	L(DDPG,Rand)
UserLAnd	0.49	0.49	0.49	0.47	-
Simple-Solitaire	0.06	0.21	0.19	0.18	L(Rand)
Authorizer	0.05	0.046	0.047	0.049	S(SAC), M(Q)
YalpStore	0.28	0.28	0.31	0.26	L(DDPG,Q,Rand)
Camera-Roll	0.26	0.37	0.25	0.25	L(Rand,DDPG,SAC)
AntennaPod	0.33	0.33	0.35	0.22	L(DDPG)
Phonograph	0.085	0.077	0.075	0.076	S(Q,DDPG,SAC)
MicroMathematics	0.17	0.17	0.30	0.18	L(DDPG,Q,Rand)
Lightning-Browser	0.28	0.28	0.36	0.29	L(DDPG,Q,Rand)
Firefox-focus	0.27	0.28	0.43	0.35	L(Rand, Q)
RedReader	0.31	0.31	0.31	0.33	-
Wikipedia	0.30	0.30	0.33	0.28	-
Slide	0.13	0.13	0.12	0.14	-

Table 4: AUCs achieved on real apps; effect size between winner and others when p-value $< \alpha$.

distracted by already seen activities or ineffective buttons (high dimensional action/observation space); (3) reach an activity located deeply in the app. Such behaviors are possible thanks to the learning capabilities of the DNNs used by Deep RL algorithms, while they are hardly achieved by the other existing approaches, including Tabular Q-Learning.

RQ7: In the presence of blocking activities or complex concatenated activities (activities with a high number of widgets or located in depth) that require the capability to reuse knowledge acquired in previous explorations, the learning capabilities of Deep RL algorithms make them the most effective and efficient exploration strategies.

6.2.1 Comparison between ARES and state-of-the-art tools

RQ8-a. Table 5 shows the coverage reached and the faults exposed by each testing tool on 68 Android apps from AndroTest. Coverage data are summarized employing boxplots in Figure 7. The highest average coverage and average number of crashes over ten runs are highlighted with a gray background. ARES achieved 54.2% coverage and detected 0.48 crashes on average. TimeMachine achieved 50.4% code coverage and 0.42 faults on average. Sapienz reached a mean code coverage of 48.8% and discovered 0.22 faults on average. Monkey achieved 43.9% code cover-

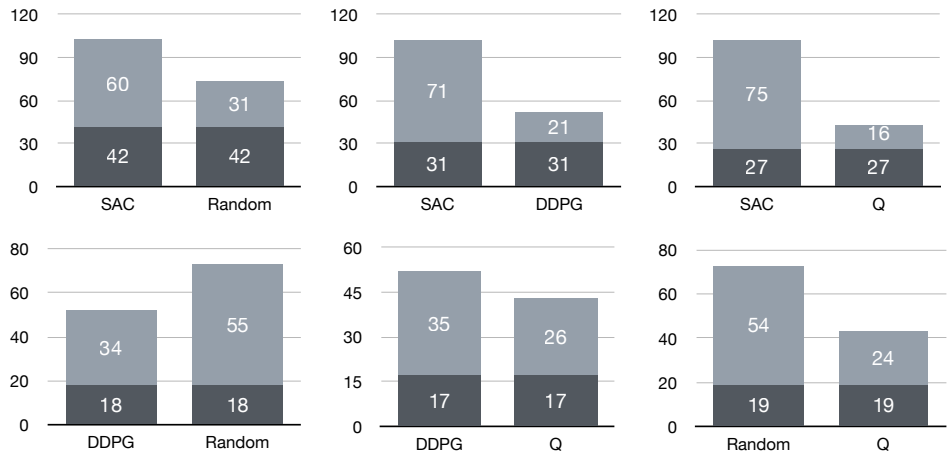

Figure 6: Comparison of total number of unique crashes on the 41 apps involved in RQ5-6: dark gray areas indicate the proportion of crashes found by both techniques.

age and discovered 0.11 faults. ARES achieved the highest code coverage on average, followed by TimeMachine, Sapienz, and Monkey. TimeMachine detected most unique crashes (179), followed by ARES (171), Sapienz (103) and Monkey (51). Actually, ARES discovered less crashes than TimeMachine mostly because TimeMachine uses a system-level event generator, taken from Stoat [45], which ARES does not support. However, system events present two major drawbacks: a) they vastly change depending on the Android version [46] [21] (despite TimeMachine is compatible with Android 7.1, it uses only system-level actions compatible with Android 4.4 [14]); and b) to execute them, root privileges are required. ARES does not require root privileges to work properly on any app (i.e., we recall that certain apps do not execute on rooted devices [48]). Analyzing the execution traces of each tool, we searched and identified the faults immediately after the generation of system events not related to the AUT. More than a third (63) of the crashes generated by TimeMachine come from system-level actions. Figure 8 shows a pairwise comparison of detected crashes among evaluated techniques. TimeMachine also finds only 20% of unique crashes found by ARES. For example, in the app *mnv*, only ARES generated a crash of the type *NullPointerException*, in which a missing control on input generates the failure of the conversion function *CharSequence.toString()*. The text field from which the bug can be generated is not immediately available, but several interaction steps with the app are required. This shows that ARES can be used together with other state-of-the-art Android testing techniques (in particular, TimeMachine) to cover more code and discover more crashes jointly.

The good results of ARES in code coverage and exposed faults are due to the reinforcement mechanisms of the RL algorithms and the reward function that drives the testing tool through states of the app leading to hard-to-reach states. Search-based techniques such as Sapienz typically observe the program behavior over an event sequence that may be very long. Hence, the associated coverage feedback, used to drive search-based exploration, does not have enough fine-grained details to support good local choices of the actions to be taken. The fitness function used by these algorithms evaluates an entire action sequence and does not consider the individual steps in the sequence. TimeMachine improved this weakness by relying on the coverage feedback obtained at an individual state to determine which portion of the app is not still explored. The drawback of this kind of approach is a higher computational cost that requires a higher testing time. In fact, while the time to dump the GUI is in common both to TimeMachine and ARES, measuring the coverage as feedback at each timestep implies three steps:

- generation of the coverage files concerning the AUT,
- retrieval of coverage files from the Android device,

- coverage computation and processing.

These steps together take on average a second and a half to be completed. The strategy of ARES relies on monitoring the transition between activities taking on average 0.1 milliseconds rather than computing the code coverage at each time step. Hence, the latter approach offers a better trade-off between the granularity of the feedback and the computational cost required to obtain it.

RQ8-a: ARES achieved the highest code coverage in 41/68 apps, followed by TimeMachine (12/68), Sapienz (6/68), and Monkey (2/68). ARES triggered crashes more often than other tools in 23/68 apps, followed by TimeMachine (15/68), Sapienz (5/68), and Monkey (0/68). However, TimeMachine generated the highest number of unique crashes (179), but 63 of them come from system-level events. ARES generates 171 faults, Sapienz 103, and Monkey 51.

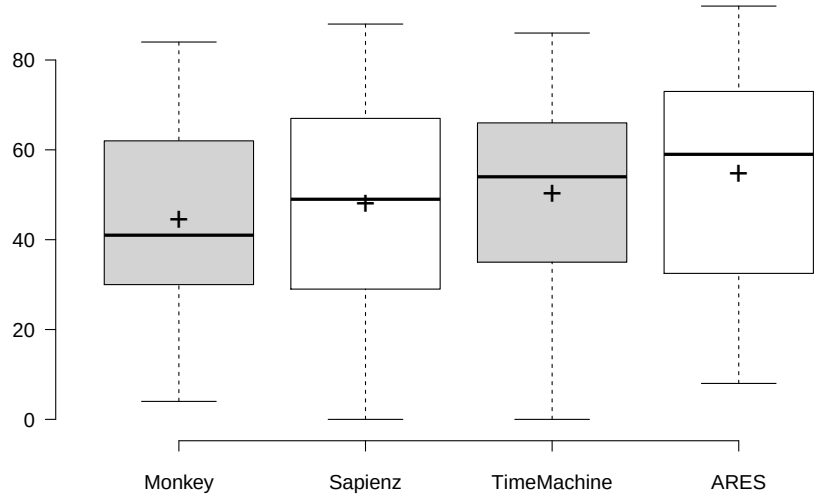

Figure 7: Code coverage achieved by ARES, TimeMachine, Sapienz, and Monkey.

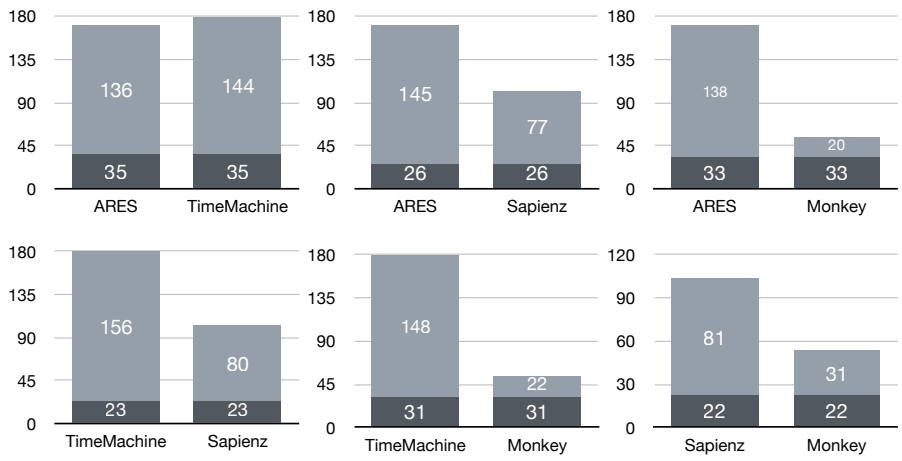

Figure 8: Comparison of total number of unique crashes involved in RQ8-a: dark gray areas indicate the proportion of crashes found by both testing tools.

App	Coverage				Faults			
	Monkey	Sapienz	TM	ARES	Monkey	Sapienz	TM	ARES
a2dp	38	44	42	42	0	0.4	0.1	0.3
aagtl	16	18	17	19	0.3	1.0	1.7	1.5
aarddict	13	15	17	17	0	0	0.2	0.2
acal	18	27	27	28	0.5	0.8	1.0	1.0
addi	19	20	17	20	0.4	0.3	0.4	0.6
adsdroid	30	36	36	35	0.1	0.4	0.5	0.7
aGrep	45	-	59	56	0.1	-	0.3	0.2
aka	65	84	77	84	0.1	0.7	0.1	0.4
alarmclock	64	41	60	71	0.5	0.5	0.6	0.8
aLogCat	67	71	75	87	0	0	0	0
Amazed	36	66	63	89	0.1	0.3	0.2	0.2
AnyCut	63	65	63	73	0	0	0	0
anymemo	31	50	43	53	0.2	0.8	0.3	1.0
autoanswer	12	16	21	15	0	0	0.5	0.5
baterrydog	63	67	62	69	0	0.1	0.4	0.5
battery	73	78	77	92	0	0.4	0.5	0.4
bites	34	41	45	43	0.1	0.2	0.9	0.5
blokish	55	52	68	45	0	0.2	0	0.3
bomber	76	73	77	84	0	0	0	0
Book-Catalogue	27	29	27	25	0.1	0.2	0.8	0.5
CountdownTimer	74	62	77	84	0	0	0	0
dalvik-explorer	66	72	70	72	0.3	0.2	0.7	0.8
dialer2	39	42	42	44	0	0	0.3	0.4
DivideAndConquer	84	83	82	80	0	0.2	1.0	0.9
fileexplorer	41	49	55	64	0	0	0	0
frozenbubble	80	76	75	70	0	0	0	0
gestures	37	52	51	55	0	0	0	0
hndroid	7	15	18	18	0.1	0.4	1.1	1.3
hotdeath	75	75	72	74	0.1	0.2	0.8	0.9
importcontacts	40	39	40	42	0.1	0	0.6	0.8
jamendo	53	41	54	63	0	0.4	1.4	1.6
k9mail	6	7	8	8	0.4	0	1.8	1.2
LNM	47	-	-	75	0	-	-	0.2
lockpatterngenerator	75	79	74	78	0	0	0	0
LolcatBuilder	26	25	29	26	0	0	0.1	0
manpages	40	73	70	74	0	0.4	0.3	0.4
mileage	38	45	48	45	0.3	1.0	2.3	1.8
Mirrored	57	59	62	59	0.4	0	0.8	0.7
mnv	41	60	43	56	0.5	0.3	1.0	1.1
multismssender	34	59	61	73	0	0.2	0.3	0.4
MunchLife	67	72	71	88	0	0	0	0
MyExpenses	41	60	50	63	0	0	0.2	0.2
myLock	25	31	50	30	0	0	0.5	0.2
Nectroid	34	66	58	57	0	1.0	0.3	0.9
netcounter	43	70	58	69	0	0	0.4	0.5
PasswordMaker	53	58	55	59	0.3	0.8	0.6	0.9
passwordmanager	7	8	17	18	0	0	0	0
Photostream	30	34	35	29	0.1	0	0.4	0.8
QuickSettings	50	45	46	52	0	0.2	0	0.4
RandomMusicPlayer	53	58	58	63	0	0	0.6	0.8
Ringdroid	22	29	48	30	0	0.1	0.3	0.2
sanity	26	19	31	22	0.2	0.3	0.5	0.4
soundboard	42	32	59	61	0	0.6	0	0.4
SpriteMethodTest	58	80	73	88	0	0	0	0
SpriteText	60	60	57	60	0	0.4	0	0.5
swiftp	12	14	13	17	0	0.4	-	0.6
SyncMyPix	21	21	23	25	0	0	0	0
tippy	75	83	74	85	0	0.4	0.3	0.4
tomdroid	47	46	51	69	0	0.3	0	0.3
Translate	49	48	48	50	0	0	0	0
Triangle	-	-	-	-	-	-	-	-
weight-chart	63	67	66	71	0	0	0	0
whohasmystuff	61	68	66	81	0.1	0	0.9	1.0
wikipedia	31	32	33	35	0	0	0	0
Wordpress	4	5	7	8	0	0.5	1.5	1.0
worldclock	83	88	86	90	0	0	0.6	0.4
yahtzee	51	57	56	69	2	0.2	0.5	0.5
zooborns	30	16	35	37	0	0	0.1	0.5
Average	43.9	48.8	50.4	54.2	0.11	0.22	0.42	0.48
Sum					51	103	179	171

Table 5: Results on 68 open-source apps coming from AndroTest.

App	Coverage		Faults	
	ARES	Q-Testing	ARES	Q-Testing
Alogcat	84	76	0	0
Antennapod	48	42	0.8	0.4
AnyCut	72	67	0	0
batterydog	65	49	0.5	0.3
Jamendo	64	46	0.4	0.7
Multismssender	71	45	0.4	0
Myexpanses	63	36	0.4	0.2
talalarmo	74	74	0.8	0.5
Tomdroid	61	50	0.3	0.2
vanilla	41	40	0.5	0.4
Average	64.3	52.5	0.41	0.27
Sum			17	16

Table 6: Comparison between Q-Testing and ARES.

RQ8-b. Table 6 shows the coverage reached and the faults exposed by ARES and Q-Testing on 10 Android apps instrumented with JaCoCo. ARES achieved 64.3% coverage and exposed 0.41 faults on average; it detected 17 unique crashes. Q-Testing achieved 52.5% code coverage and 0.27 crashes on average, and it detected 16 unique crashes. ARES achieved the highest code coverage on almost all apps, and on average ARES covered 12% more code than Q-Testing. Q-Testing generated six faults in common with ARES, while the other four faults are generated using the system-level events of Stoat. In the app *antennapod*, only ARES generated a *NumberFormatException* with a text field located deeply in the Settings submenu. In this comparison, the main advantage of ARES seems to be a better reward function that encourages the tool to visit the greatest number of activities within the same episode. Instead, Q-Testing determines the reward of the Q-Learning algorithm by computing the similarity between Android app states, which does not guarantee an efficient way to overcome blocking activities.

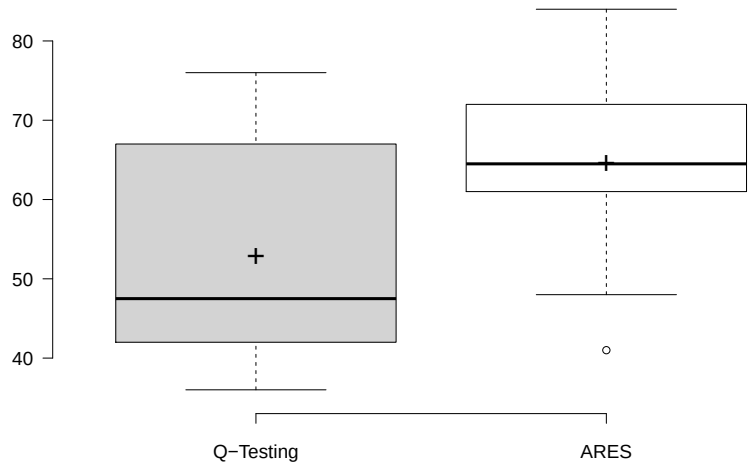

Figure 9: Code coverage achieved by ARES and Q-Testing.

RQ8-b: ARES reached the highest code coverage on 9/10 apps, the highest average of crashes in 7/10 apps, and the highest number of unique crashes. Q-Testing, the state-of-the-art RL Android testing tool, reached the same level of exploration of ARES only in 1/10 app, and the highest average number of crashes on only 1/10 apps.

6.3 Threats To Validity

We adopted several strategies to enhance the internal validity of our results. We chose apps coming from a standard testing benchmark used in previous studies to mitigate risks of selection bias. We used default settings, given the same starting condition, and ran each tool several times under the same workload to ensure that no testing tool was at a disadvantage. We followed the Stoat protocol to identify unique crashes, and also we manually checked the crashes found. To measure coverage, we used JaCoCo, a standard coverage tool.

7 Implications and Limitations

Using ARES as testing tool involves several benefits:

- **black-box automated testing:** ARES relies only on the GUI of the AUT. This allows developers to test their production apps with no modifications. Available state-of-the-art tools, such as Sapienz and TimeMachine, rely on code coverage to drive exploration, and as recognized by the researchers who developed them, this makes testing less efficient.
- **wide compatibility:** ARES works on Windows, Linux and MacOS. ARES can test apps in parallel on emulators or real devices with Android from 6.0 to 10.0.
- **policy reuse:** at the end of the testing phase, ARES saves the status of the neural network as a policy file. Instead of restarting ARES from scratch each time a new version of the AUT is launched, the policy can be loaded and reused on a later version of the AUT.
- **modularity:** within ARES, the app environment is decoupled from the RL algorithm used during the testing phase, allowing to deploy new algorithms easily.

Despite the advantages given by ARES there are also some limitations:

- **benefits on easy apps:** performance on simpler apps sometimes align with the performance of dummy methods such as random algorithms and does not justify the use of Deep RL.
- **system-level events:** ARES implements a limited set of events that act at the system-level, including the most commonly supported ones, such as *toggle Internet connection* and *rotate screen*. The other system-level events require rooted devices to work, which brings some drawbacks, among which the inability to work with apps that, for security reasons, perform a "root-check" and stop working if the device is rooted.

8 Related Work

As suggested by [28], existing research on Android testing can be classified by the methodologies that the testing approaches adopt.

8.1 Random Testing

Testing tools in this category generate random events on the AUT GUI. Monkey [22] is one of the most popular black-box Android testing tools. It triggers events by interacting randomly with screen coordinates. This simple random approach worked relatively well on some benchmark applications [12]. Nonetheless, Monkey tests involve many ineffective or repeated events, as there is no guidance to make the exploration efficient. ARES implements a smarter version of Monkey, which we used as a baseline (Random). Such an improved version selects only actions that are possible in a given GUI state, thus making the exploration strategy a bit more efficient.

8.2 Model-based Testing

Model-based tools [5] [6] [24] first build navigation models of the Android app by means of static or dynamic analysis, used to explore efficiently the application states, and then they extract test cases from such models, to eventually expose bugs. AndroidRipper [5], MobiGUITAR [6] try to maximize the exploration by using the ripping technique. Guo et al. [24] use static analysis to effectively improve GUI exploration performance. Stoat [45] uses a stochastic FSM to model the app behavior. The app model is built using dynamic analysis, enhanced with a weighted UI exploration strategy, and with the help of static analysis. Compared to Stoat, ARES can be viewed as computing a navigation model implicitly: the MDP model used by the Deep RL algorithms. One key advantage of using an implicit model is that we do not have to deal with the combinatorial explosion of its size, which Stoat controls using model compaction heuristics.

8.3 Structural Testing

Structural strategies [7, 17, 36, 15, 37] generate coverage oriented inputs using symbolic execution or evolutionary algorithms. Sapienz [37] maximizes code coverage and bug revelation using a Pareto-optimal multi-objective search-based approach, which applies genetic operators such as mutation and crossover to produce new test cases. It can generate specific input for text fields by reverse-engineering the APK. This process occasionally results in invalid sequences discarded by the fitness functions that reward test cases with high coverage. TimeMachine [15] improves Sapienz by identifying interesting states in the past and restarting the search process from them when the search stagnates. While coverage-oriented approaches require the possibility to instrument the app to measure coverage and possibly execute it symbolically, our approach is black-box and does not suffer from the scalability issues affecting, e.g., symbolic execution.

8.4 Machine Learning Based Testing

Some ML-based testing approaches [8] [29] [31] use an explicit, supervised training process to learn from previous test executions. They can reuse previous knowledge acquired on different apps or past versions of the app under test. Approaches as QBE [29] make the transfer of knowledge to new apps possible by abstracting the app state in a form that is supposed to hold across different domains and implementations. However, the effectiveness of such a transfer learning process depends on the similarity between new and old apps.

One of the first works proposing RL for GUI testing is AutoBlackTest [39]. This approach is based on the simplest form of RL, Tabular Q-Learning, whose effectiveness is strongly dependent on the initial values in the Q-Table. On the contrary, ARES learns the action-value function from scratch during the exploration of the AUT. One of the most recent approaches to Android testing based on Deep Learning is Q-Testing [42]. However, it also uses Tabular Q-Learning as a backbone. At the same time, learning is limited to the computation of the similarity between Android app states, which determines the reward of the Q-Learning algorithm. ARES instead learns both the state similarity and the action-value function during its interactions with the AUT.

9 Conclusion and Future Work

We have proposed an approach based on Deep RL for the automated exploration of Android apps. The best exploration strategy is learned automatically as the test progresses. The approach is implemented in the open-source tool ARES, which is complemented by FATE, a model-based Android testing tool that we developed to support fast execution and configuration of the alternative Deep RL algorithms of ARES. The resulting configuration of ARES, particularly when running the

SAC Deep RL algorithm, outperformed all the considered baselines in terms of coverage achieved over time and exposed bugs.

In our future work, we plan to investigate specific fault categories that are particularly relevant for Android apps, such as security vulnerabilities. In fact, we think that the adaptation and reward mechanisms used by Deep RL algorithms to learn the optimal exploration strategy could be particularly effective when the fault to be exposed is a security fault. We also plan to port ARES and FATE to iOS.

References

[1] Emma, 2006.

[2] Appbrain, 2020.

[3] Appium, 2020.

[4] Josh Achiam. Key concepts in rl, 2018.

[5] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 258–261. IEEE, 2012.

[6] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. Mobiguitar: Automated model-based testing of mobile apps. IEEE software, 32(5):53–59, 2014.

[7] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pages 1–11, 2012.

[8] Nataniel P Borges, Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pages 133–143. IEEE, 2018.

[9] Justin A Boyan and Andrew W Moore. Generalization in reinforcement learning: Safely approximating the value function. In Advances in neural information processing systems, pages 369–376, 1995.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. arXiv preprint arXiv:1606.01540, 2016.

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

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

[13] Jack Clark and Dario Amodei. Faulty reward functions in the wild, 2016.

[14] Zhen Dong, Marcel Bohme, Lucia Cojocaru, and Abhik Roychoudhury. Github repository: Timemachine, 2020.

[15] Zhen Dong, Marcel Bohme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In Proceedings of the 42nd International Conference on Software Engineering (ICSE), pages 481–492, 2020.

[16] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. arXiv preprint arXiv:1802.09477, 2018.

[17] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. Android testing via synthetic symbolic execution. In 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 419–429. IEEE, 2018.

[18] Google. Broadcasts, 2019.

[19] Google. Safety net, 2019.

[20] Google. Android emulator, 2020.

[21] Google. System-level events api 25, 2020.

[22] Google. Ui/application exerciser monkey, 2020.

[23] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 269–280. IEEE, 2019.

[24] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. Improving automated gui exploration of android apps via static dependency analysis. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 557–568, 2020.

[25] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. arXiv preprint arXiv:1801.01290, 2018.

[26] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[27] Sture Holm. A simple sequentially rejective multiple test procedure. Scandinavian journal of statistics, pages 65–70, 1979.

[28] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. IEEE Transactions on Reliability, 68(1):45–66, 2018.

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

[30] Edward A. Lee. Finite state machines and modal models in ptolemy ii. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.

[31] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: a deep learning-based approach to automated black-box android app testing. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1070–1073. IEEE, 2019.

[32] Yuxi Li. Deep reinforcement learning: An overview. arXiv preprint arXiv:1701.07274, 2017.

[33] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[34] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[35] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 224–234, 2013.

[36] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of android apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 599–609, 2014.

[37] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, pages 94–105, 2016.

[38] Evgeny Mandrikov Marc R. Hoffmann, Brock Janiczak. Jacoco code coverage, 2020.

[39] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pages 81–90. IEEE, 2012.

[40] Microsoft. your device is rooted and you can't connect-android, 2020.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[42] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 153–164, 2020.

[43] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In European Conference on Machine Learning, pages 317–328. Springer, 2005.

[44] et al. Silver, David. Deterministic policy gradient algorithms. 2014.

[45] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pages 245–256, 2017.

[46] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. System-level events api 19, 2017.

[47] Sutton. Reinforcement Learning: An Introduction. 2014.

[48] Vincent F Taylor and Ivan Martinovic. Short paper: A longitudinal study of financial apps in the google play store. In International Conference on Financial Cryptography and Data Security, pages 302–309. Springer, 2017.

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

10 Appendix

10.1 Study 1- Deep RL

Certain parameters of the algorithms are omitted for simplicity, and can be found in the documentation of Stable Baselines [26].

Control Policy:

Deep RL algorithms rely on a policy based on a MLP composed of 2 layers and 64 neurons each.

Learning rates:

- DDPG: 0.0001
- SAC: 0.0003
- TD3: 0.0003

DDPG Configuration	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
random_exploration	0.5	0.5	0.6	0.6	0.7	0.7	0.8	0.8
nb_train_steps	5	25	5	25	5	25	5	25

TD3 Configuration	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
random_exploration	0.5	0.5	0.6	0.6	0.7	0.7	0.8	0.8
train_frequency	25	100	25	100	25	100	25	100

SAC Configuration	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
target_update_interval	1	1	2	2	5	5	10	10
train_frequency	1	5	1	5	1	5	1	5

10.2 Study 1- Tabular RL

10.3 Study 2

Some parameters used in Study 2 have been selected from the results of Study 1 (see Section 6).

DDPG:

- Control Policy: MLP, 2 layers, 64 neurons
- Learning rate: 0.0001
- nb_train_steps: 10
- random_exploration: 0.7

SAC:

- Control Policy: MLP, 2 layers, 64 neurons
- Learning rate: 0.0003
- train_freq: 5

Q-Learning Configuration	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
epsilon	0.5	0.6	0.7	0.8	0.5	0.6	0.7	0.8
gamma	0.99	0.99	0.99	0.99	0.9	0.9	0.9	0.9
alpha	0.628	0.628	0.628	0.628	0.628	0.628	0.628	0.628

- target_update_interval: 10

TD3:

- Control Policy: MLP, 2 layers, 64 neurons
- Learning rate: 0.0003
- train_freq: 10
- random_exploration: 0.8

Q-Learning:

- epsilon: 0.8
- gamma: 0.9
- alpha: 0.628