Reinforcement Learning Based Curiosity-Driven Testing of Android Applications

Minxue Pan*
mxp@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Software Institute, Nanjing University
Nanjing, China

An Huang
ha@smail.nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China

Guoxin Wang
njuwgx@smail.nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China

Tian Zhang*
ztluck@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China

Xuandong Li
lxd@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China

ABSTRACT

Mobile applications play an important role in our daily life, while it still remains a challenge to guarantee their correctness. Model-based and systematic approaches have been applied to Android GUI testing. However, they do not show significant advantages over random approaches because of limitations such as imprecise models and poor scalability. In this paper, we propose Q-testing, a reinforcement learning based approach which benefits from both random and model-based approaches to automated testing of Android applications. Q-testing explores the Android apps with a curiosity-driven strategy that utilizes a memory set to record part of previously visited states and guides the testing towards unfamiliar functionalities. A state comparison module, which is a neural network trained by plenty of collected samples, is novelly employed to divide different states at the granularity of functional scenarios. It can determine the reinforcement learning reward in Q-testing and help the curiosity-driven strategy explore different functionalities efficiently. We conduct experiments on 50 open-source applications where Q-testing outperforms the state-of-the-art and state-of-practice Android GUI testing tools in terms of code coverage and fault detection. So far, 22 of our reported faults have been confirmed, among which 7 have been fixed.

*Corresponding authors.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397354

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Android app testing, reinforcement learning, functional scenario division

ACM Reference Format:
Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397354

1 INTRODUCTION

The proliferation of mobile devices and apps makes a huge impact on our daily life. Statistics [50] show that an average user spends more than 2 hours a day on mobile apps. However, it remains a challenge to guarantee apps' quality because of the large combinational space of possible events and transitions. A daily used app usually contains plenty of interfaces and executable events, which makes it time-consuming to explore all its states, let alone particular functionalities that can be accessed only in specific conditions. Different strategies have been applied to automated testing of Android applications, which, unfortunately, still need improvement [10].

Random strategies [21, 31] generate pseudo-random events to fuzz the application under test. Monkey [21], regarded as the state-of-practice testing tool, is a typical example of this strategy. Despite the wide adoption in practical development, its shortcomings are quite obvious. It is common for Monkey to generate futile events like clicking a non-interactive area of the screen that makes no changes to the current state. Also, the testing is unbalanced and some hard-to-reach functionalities may never be explored.

Model-based strategies [2, 5, 46] generate test cases according to application models which are constructed with a static or dynamic

approach. In this case, high-quality models are extremely important in order to achieve a good testing result. However, as illustrated above, it is challenging to explore all the states of an Android application. What's more, it is almost impossible to precisely model the apps' behavior. For example, a welcome interface is common in nowadays applications which will appear only when the app is opened the first time. This information often cannot be captured, making the generated event sequences always contain events in the welcome interface, which is inconsistent with apps' actual behavior and is highly detrimental to the testing effectiveness.

Systematic strategies [4, 27, 32] use sophisticated techniques such as symbolic execution to provide specific inputs for targeted application behavior. These strategies are mainly designed to reveal typical functionalities that are hard to execute with other strategies, but they are less scalable and often perform worse in overall testing metrics like code coverage and bug revelation.

Machine learning techniques are starting to find application in Android GUI testing, too. Recently, a few works [1, 26, 48] utilize reinforcement learning, specifically Q-learning which can benefit from both random and model-based approaches, to guide the testing progress. The Q-table, which records each event's value, along with the propagation property of Q values can partly take the place of models to store testing related information in a light way. It also helps to avoid the inconsistency problem between models and apps' actual behavior which is an advantage of random testing. However, existing works do not fully unleash the ability of reinforcement learning in Android testing. Take the reward giving process, which is key to reinforcement learning, as an example. Prior works tend to calculate the differences between two states (the ones before and after an event is executed) to determine the reward. If two states are quite different, the testing tools will continuously give a large reward which results in frequently jumping between them even if they have been over explored. Although some of the reward calculating functions take the executing frequency into consideration, the problem arises again when most of the events have been executed several times.

To tackle the aforementioned challenges and to unleash machine learning's potential in Android GUI testing, we propose a novel approach, Q-testing, based on reinforcement learning. The strategy of Q-testing is called curiosity-driven exploration which guides testing towards states that it is curious about. More precisely, Q-testing maintains a set, which acts as memory, to record part of the previously visited states. The reinforcement learning reward is calculated according to the differences between the current state and those recorded in memory. Different from prior works, curiosity-driven exploration is a dynamically adaptive strategy. By adaptive, we mean that it can spot changes in the importance of states and will continuously adjust the reward for a certain event.

In order to improve test efficiency, we novelly propose a neural network to divide different states at the granularity of functional scenarios. Q-testing uses this module to compare states and calculate rewards. With its help, Q-testing will preferentially make effort to covering different functionalities which helps to rationally allocate limited testing time and will result in the rapid growth of code coverage in a short time. We collect more than 6k samples to train the model and it can be used not only in reinforcement learning. Other tasks including state compression and code recommendation may also benefit from it.

Q-testing also involves testing strategies that are specifically designed for Android applications. We resolve RecyclerView [19] and ListView [18] in a manner to forbid the waste of time in unnecessary testing. We also take system-level events into consideration and it helps Q-testing find some complicated bugs.

In this paper, we make the following main contributions:

- We propose a novel curiosity-driven exploration strategy based on reinforcement learning to guide Android automated testing.
- We collect samples and train a neural network, which can effectively divide different states at the functional level in an efficient manner.
- We implement a tool and conduct a large-scale experiment. Results show that our approach outperforms existing ones in terms of both code coverage and fault detection. The tool and experimental data are publicly available[1] to facilitate future researches.

The remainder of the paper is structured as follows. Section 2 gives an introduction to Q-learning. Section 3 describes our reinforcement learning based testing strategy. Section 4 presents the state comparison module. Section 5 presents the experiment results. Section 6 surveys related work and Section 7 makes a conclusion.

2 Q-LEARNING

Q-learning [49] is a form of model-free reinforcement learning whose goal is learning how to map situations to actions so as to maximize a numerical reward signal in an unknown environment. The two most distinguishing features of reinforcement learning are trial-and-error search and delayed reward. The agent must try different actions to discover which may yield the most reward, and actions may affect not only the immediate reward but also subsequent states along with future rewards.

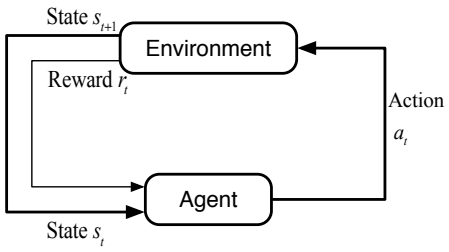


Figure 1: Markov Decision Process

The problem of reinforcement learning can be formalized with ideas from dynamic systems theory, specifically the Markov decision process, or MDP. MDP can be defined as a 4-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$ where $\mathcal{S}$ represents the set of states and $\mathcal{A}$ represents the set of actions. As depicted in Figure 1, the agent and the outside environment interact at each of the discrete time steps of a sequence. At each time step t, the agent selects and executes an action, $a_t \in \mathcal{A}$,

[1] https://github.com/anlalalu/Q-testing

based on its observation of the state, $s_t \in \mathcal{S}$. After that, the agent will move to a new state, $s_{t+1} \in \mathcal{S}$, and receive an immediate reward, $r_t \in \mathcal{R}$, at the same time. The variables r_t and s_t have well-defined probability distributions dependent only on preceding state and action. That is, $s_{t+1} \sim \mathcal{P}(s_t, a_t)$ and $r_t \sim \mathcal{R}(s_t, a_t)$. The return which represents the cumulative reward is often defined as $R_t = \sum_{t>0} \gamma^{t-1} r_t$, where future rewards are discounted by a factor of $\gamma \in [0, 1]$.

Reinforcement learning usually involves Q value functions to estimate how good it is to perform an action in a certain state. This action-value function returns the expected cumulative reward of a sequence of actions which starts from action A_t in state S_t and thereafter following policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}\left[R_t | s_t, a_t\right] \tag{1}$$

With the Bellman equation, we can express the relationship between the value of a state-action pair and its successor state-action pair:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}\left[r(s_t, a_t) + \gamma Q^{\pi}(s_{t+1}, \pi_{s_{t+1}})\right] \tag{2}$$

Q-learning uses equation 2 to estimate each state-action pair. If the states and actions are discrete and finite, the pairs can be represented in a tabular form where all pairs will be given an initial value. Every time an action is executed, the relating state-action value will be updated:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q^{*}(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{3}$$

$Q^{*}(s_{t+1}, a_{t+1})$ represents the maximum cumulative reward that can be achieved from state s_{t+1}. The α is the learning rate which is between 0 and 1. As we can see in this equation, the value of subsequent state-action pair will be propagated to influence preceding pairs. There is a strict proof that the estimator will converge to the true value if the environment is explored sufficiently. Whenever Q-learning estimates the values precisely, we can easily find the optimal policy simply by executing the action which has the highest value at every state.

Q-learning provides agents with the capability of learning to act optimally in Markovian environments without requiring them to build models of the environments. We observe that an Android application testing process can be viewed as a Markov Decision Process: by giving rewards when test actions lead to new states of applications, the entire testing should be able to learn to cover more functionalities of the applications. This inspires us to apply Q-learning to Android automated testing.

3 Q-LEARNING BASED ANDROID TESTING

The Android testing task can be viewed as a Markov Decision Process which makes it possible for reinforcement learning to play a role. We design exploration strategies based on Q-learning to guide the testing tool towards unrevealed and unfamiliar functionalities.

3.1 Approach Overview

The workflow of Q-testing is depicted in Figure 2. Analogs to the process of MDP, Q-testing interacts with the outer environment, the application under test (AUT), during testing. In each cycle, Q-testing

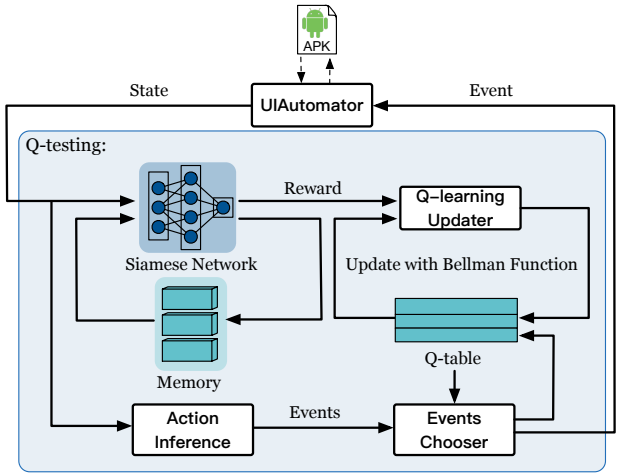


Figure 2: Q-testing Workflow

first observes the application's current state s_t with UIAutomator [20]. Then the state s_t is compared with part of prior observed states. The states are stored in a buffer which acts like Q-testing's memory. A neural network is trained to extract states' features and conduct the comparison. If state s_t is similar to any of the states in memory, the comparator will give a small reward. Otherwise, the module will give a large reward and state s_t will be added to the memory buffer. The reward is used to update the value of the state-action pair (s_{t-1}, a_{t-1}). All the state-action pairs' values are stored in Q-table and their values are updated with equation 3. Every time a new state is reached, Q-testing will add related state-action pairs into Q-table and initialize it with a large value to encourage the execution of new events. After the updating of Q value, Q-testing will infer the executable events from the GUI hierarchy of s_t and choose an event a_t with reference to Q-table. In most instances, the event with the highest value will be chosen. After the execution, the AUT will respond to a_t and another cycle starts.

3.2 Formulating Android Testing as MDP

In our approach, an Android GUI testing problem is mathematically formalized as an MDP that can be defined with a 4-tuple, $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$. How we define $\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$and $\mathcal{R}$ for the Android GUI testing task will make an impact on the testing effectiveness and efficiency. In this paper, we adopt the following formulation.

$\mathcal{S}$: States. Prior Android automated testing works [1–3, 5, 6, 23, 46, 48, 53] adopt different criteria to abstract applications' states. Baek et al. [6] conducts experiments to identify the effect of comparison criteria on Android GUI testing and the result indicates that finer comparison granularity will benefit testing by achieving higher code coverage and finding more bugs. For this reason, we adopt the widget composition as a comparison criterion. Specifically, Q-testing uses UIAutomator to extract the GUI hierarchy. UIAutomator dumps the information of the widgets that are contained in the current interface. The widgets are arranged in a tree structure where non-leaf nodes represent layout widgets and leaf nodes represent executable widgets. Q-testing ignores some of the widgets' attributes including text to avoid state explosion. This

detailed information will be disruptive to the update of Q-table and will reduce testing efficiency. In brief, our state s_t is defined as a combined state $(w_1, w_2, \ldots w_n)$ where w_i is the state of widget contained in s_t and it is defined by selected attributes which include an index attribute to describe the widget's position in the GUI hierarchy tree.

$\mathcal{A}$: Actions. We formulate user interaction events in apps as actions in MDP, and we do not distinguish events and actions in this paper. Similar to previous work [6, 23, 46], Q-testing infers executable events in the current state by analyzing the dumped widget hierarchy and corresponding attributes (e.g., clickable, scrollable).

As each event is associated with a specific state, this enables us to also use the state-action pairs to represent an event executable in a state of the application. Equation 4 describes the ϵ-greedy policy that is utilized by Q-testing to select the next event. Q-testing selects the event with the highest Q value with probability $1-\epsilon$ and selects a random event with probability ϵ. In order to trigger intricate bugs, Q-testing also takes system-level events into consideration. The value of ϵ can be adjusted and its default setting is 0.2 in Q-testing.

$$getAction(s) = \begin{cases} \arg\max_a Q(s,a), & 1-\epsilon \\ \text{random UI event}, & \frac{1}{2}\epsilon \\ \text{random system event}, & \frac{1}{2}\epsilon \end{cases} \quad (4)$$

P: Transition Function. The transition function depicts which state the application will transit to after an event is executed. It is determined by the AUT and we cannot make any change to it. What should be emphasized is the non-deterministic transition of which the result of executing an event can be changed by the states of internal variables. Prior model-based approaches usually ignore this kind of transition due to the limited ability of their modeling processes. Q-testing updates Q values with the Bellman function of which the value of a state-action pair (s, a) is influenced by all the states that can be reached from s. Therefore, all the transition information is considered when the values are updated.

R: Reward. Q-testing receives a reward every time it executes an event. We propose a policy to determine the reward, which contributes to our exploration strategy. The details are illustrated in the following subsections.

3.3 Exploration Strategies

Q-testing employs the curiosity-driven strategy to test the AUT. Curiosity, which encourages the agent to explore the outer environment, is studied in reinforcement learning tasks [9, 25, 38, 43] to solve the problem of sparse reward. The curiosity-driven strategy proposed by Q-testing can guide the testing tool to explore unfamiliar states in order to cover codes and find bugs with high efficiency.

Algorithm 1 describes the algorithm of the curiosity-driven exploration strategy. Specifically, Q-testing maintains a state buffer to store part of previously visited states (line 2). This buffer acts like a memory and it helps Q-testing to find out the states that it is unfamiliar with, in other words, curious about. Q-table stores all the state-action pairs grouped by activities (denoted as *act*) along with their values. Every time a state is reached, Q-testing will directly look up the Q-table to find out the executable actions. If the state is reached for the first time, Q-testing will infer executable actions

Algorithm 1 Curiosity Driven Testing

Input: the app under test AUT, execution time t, learning rate α, similarity threshold $threshold$

1: $Q \leftarrow \emptyset$ ▷ initialize Q-table
2: $M \leftarrow \emptyset$ ▷ initialize memory buffer
3: $s_{t+1}, act_{t+1} \leftarrow$ getCurrentState(AUT)
4: $v_{t+1} \leftarrow$ extractVector(s_{t+1})
5: $M \leftarrow M \cup (act_{t+1}, v_{t+1})$
6: **while** ¬timeout(t) **do**
7: $s_t \leftarrow s_{t+1}$
8: $A \leftarrow$ getOrInferEvents(Q, s_t)
9: $a_t \leftarrow$ getAction(Q, s_t)
10: $s_{t+1}, act_{t+1} \leftarrow$ execute(AUT, a_t)
11: $v_{t+1} \leftarrow$ extractVector(s_{t+1})
12: **for all** $(act_{t+1}, v) \in M$ **do**
13: $d \leftarrow$ calculateDistance(v, v_{t+1})
14: $similarity \leftarrow min(similarity, d)$
15: **end for**
16: **if** $similarity >= threshold$ **then**
17: $r_t \leftarrow smallReward$
18: **else**
19: $M \leftarrow M \cup (act_{t+1}, v_{t+1})$
20: $r_t \leftarrow largeReward$
21: **end if**
22: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q^*(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$
23: **end while**

from the GUI hierarchy and initializes them in Q-table (line 8). The initial value for all the unexecuted actions is set to 1000, which is a large number, to encourage executing of new actions and it will be assigned to 0 after executed. Q-table acts like a lightweight model that speeds up the testing process, and what is more, it avoids the problems caused by imprecisely modeling. Q-table also guides the selection of next executed action as depicted in equation 4 (line 9). After an action a_t is executed from state s_t, the AUT will response to it and transit to a state s_{t+1} (line 10). Then s_{t+1} is compared with those stored in memory and a reward will be given according to their differences (lines 11-21). The curiosity-driven strategy will steer testing towards unfamiliar states by giving a large reward when reaching a state that is, to a certain extent, different from all the remembered states. Q-testing will also update its memory by adding the unfamiliar states into the buffer and the next time the state is visited, it will not be attractive anymore. We set the reward to be 500 for actions leading to unfamiliar functionalities, which will improve the action's value according to the Bellman equation. As for actions that make little contribution to Q-testing exploring new parts of the AUT, the reward is set as -500. The reward values are configurable, however, these default values work well on different types of applications, as shown in the experiments.

It is worth emphasizing that the strategy benefits from the propagation property of Q value (line 22), which makes it possible to guide Q-testing towards valuable states no matter what values of the current state is. Suppose that Q-testing executes an event a_t and the AUT transits to state s_{t+1} that is similar to one of the states in memory set. Then a -500 reward will be given which may decrease the value of a_t. However, if s_{t+1} is a valuable state that can lead

to unexplored scenarios, then $Q^*(s_{t+1}, a_{t+1})$, which is the maximum of the Q-values of all state-action pairs in state s_{t+1} stored in the Q-table, will be large. This results in that the value of a_t will still be improved, since γ is set to 0.99 in our implementation to strengthen the impacts of new states. This makes the unexplored functionalities more likely to be explored.

Additionally, since the current state has to be compared with all those in memory, several measures are taken to improve the comparing efficiency. First, Q-testing stores feature vectors rather than the GUI hierarchy information in memory (line 11, 19). Similarity can be computed between two vectors with Manhattan distance function without extracting features repeatedly. Second, the vectors are grouped by activities and a vector only has to be compared with those in the same activity (line 12). Third, We propose a coarse-grained criterion to divide states and it benefits the testing process in several ways. On the one hand, it restricts the number of states stored and further reduces time spent in comparing states. On the other hand, it encourages Q-testing to explore states with much more differences and this will result in high code coverage in a short time since the states with a huge difference are often bound to different codes. Other details of this comparison criterion will be described in section 4.

3.4 Android-Specific Strategies

Despite the curiosity-based strategy, Q-testing also takes advantage of other strategies specifically designed based on characteristics of Android applications, including the special treatment of special widgets and the injection of system events.

RecyclerView [19] and ListView [18] are two Android widgets that are able to display a collection of views in a scrollable manner. In most cases, these widgets will contain a lot of items and users can continuously view new items by scrolling. The clicking of these items often leads to similar screens (with the same GUI hierarchy and different contents) and can only trigger the same codes. Existing works make no consideration on these widgets and may waste plenty of time testing different items. Q-testing analyzes the result of relative events. If several items in a RecyclerView lead to similar states, Q-testing will ignore the items except the first one. This strategy can save a lot of time since RecyclerView and ListView are quite common in our daily used applications.

System events can benefit testing by triggering intricate bugs. Similar to Stoat [46], Q-testing takes three kinds of system-level events, including user actions (e.g., screen rotation, phone calls), broadcast messages [17] (e.g., switch into or out of airplane mode) and application-specific events which can be extracted from Android manifest files, into consideration. Q-testing executes system-level events randomly during its exploration. These events are not included in Q-table since the number of possible system events is very large. Once they are added, Q-testing may try to execute all of them in every state.

4 SCENARIO DIVISION MODULE

We propose a new state comparison criterion at coarse granularity and to determine the reward in the curiosity-driven exploration strategy. The module is able to determine whether two states are in the same functional scenario. Reward determined by the comparison module will guide Q-testing to firstly cover different functional scenarios. The coarse granularity states division helps to allocate limited testing time rationally by preventing Q-testing from falling into the several same scenarios at the beginning. This will also result in high code coverage in a short period of time since different scenarios are usually bound to different codes. More specifically, a great number of codes (e.g., the codes used to initialize the layout and interaction events) may be covered the first time a functional scenario is reached, which is determined by the GUI driven feature of Android framework.

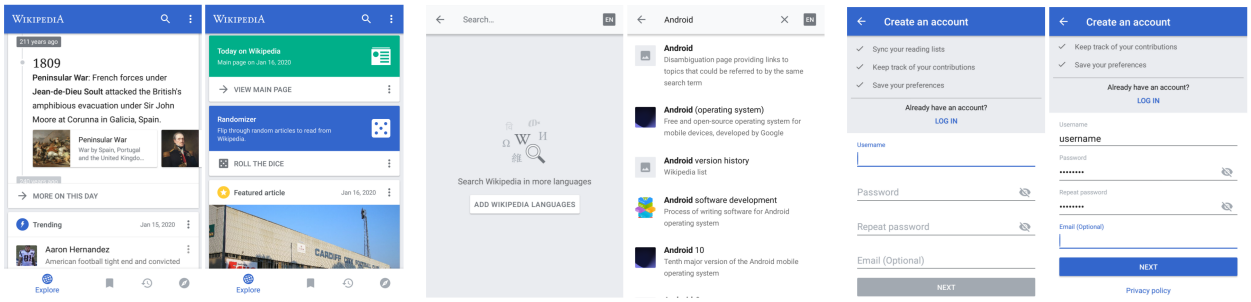
4.1 The Task of Scenario Division

The definition of a scenario for our scenario division module is similar to the definition of the use case which describes how users will perform the tasks, i.e., an outline of an application's behavior from a user's point of view. They describe the functions provided by the application without drilling into details. However, since different users have different understandings and a function may contain some sub-functions, the division of scenarios can be different. For example, the function of reading news can be a use case for a news app, but it can also be divided into two scenarios which include generally browsing a list of news and reading the details of one piece of news. However, this would not cause problems for our scenario division module. As we employ a neural network based learning framework, the granularity of dividing scenarios can be adjusted by feeding different training data.

Figure 3 shows three typical scenarios that are common in Android applications, and each scenario consists of two GUI states. As we can see, the states that are in the same scenario can be quite different and it will be difficult for existing state comparison criteria to make a correct division. Existing researches rely on manual defined features (e.g., executable events, GUI hierarchy, activity name) to describe the states. Some of them utilize a threshold to make the division flexible (e.g., two interfaces with 80% same widgets will be judged as the same state). However, prior approaches cannot meet our requirement where more properties should be considered and some of them are quite hard for a human to extract. Take Figure 3(a) as an example. There are two states in the browsing scenario which contains a RecyclerView to present items of news. These two states can access each other by scrolling. In order to successfully determine that the two states are in the same scenario, firstly, the number of items should be ignored. Second, the items' GUI hierarchies should not be viewed as equally important as other widgets. Despite the heavy work of feature selection, it is also a difficult task to manually figure out a probable threshold as it usually relies on a large amount of data and experiment. Q-testing tackles these problems under the help of the neural network which excels at extracting features.

4.2 Siamese LSTM

Q-testing utilizes the siamese network [8] to measure the similarity between two states from the perspective of judging whether they are in the same scenario. The siamese network is able to learn an invariant and selective representation through information about the similarity between sample pairs. It has been applied in NLP (Natural



(a) Browsing Scenario (b) Searching Scenario (c) Signing Up Scenario

Figure 3: Examples of Scenarios in Android Applications

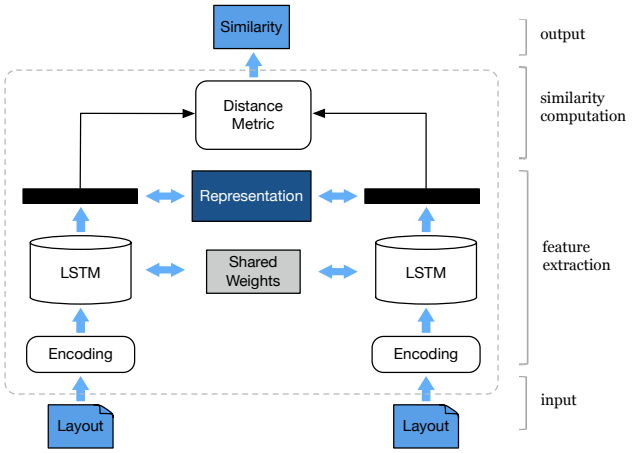


Figure 4: Architecture of Scenario Division Module

Language Processing) to learn the similarity between words [37] or sentences [36] whose goals share a lot in common with ours.

Figure 4 shows an overview of the network architecture in Q-testing. Each training sample of our task can be denoted as (s_a, s_b, y) where s_a and s_b are layout files dumped by UIAutomator. Label $y \in 0, 1$ depicts whether the two states can be divided into the same scenario ($y = 1$) or not ($y = 0$).

Layout files (s_a, s_b) which are in the form of XML will firstly be encoded as sequences $(s_1^{(a)}, s_2^{(a)}, ..., s_{l_a}^{(a)})$ and $(s_1^{(b)}, s_2^{(b)}, ..., s_{l_b}^{(b)})$ where s_i denotes the vector representation of a node in XML which contains information of a GUI widget. Q-testing utilizes different encoding strategies for different types of widget properties:

- ***Integer.*** The bounds attribute which contains 4 integers is used to demonstrate the position of a widget on the screen (2 integers to denote the position of the left top point and the other 2 denote the position of the right bottom point). Q-testing normalizes these values with the width and length of the screen.
- ***Boolean.*** The events related attributes (e.g., clickable, scrollable) often use boolean values to describe the executable properties of a widget. Q-testing maps true and false into 1 and 0 to encode such attributes.
- ***String.*** Attributes such as resource id and text are represented in the form of string. When comparing two states, we only care about whether 2 strings are the same rather than what their actual values are. Q-testing hashes the texts into numbers with the MD5 message-digest algorithm.
- ***String (Class Type).*** The class attribute describes the widget's class type and it is in the form of string too. Q-testing processes this attribute with one-hot encoding since all the types of Android widgets can be accessed.

The encoded sequences will then be passed to the dual-LSTM (Long Short-Term Memory) network which comprises two single-layer networks with 100 hidden neurons. The neural network is able to learn a mapping from the space of variable length sequences into a fixed length vector. Finally, the distance of the generated vectors will be measured with similarity functions like Manhattan distance and the weights of the layers in LSTMs will be updated according to the difference between output and actual label. Note that the LSTMs share the same weights, they will extract features with the same standard.

After the training process, part of the model is used in the scenario division module. Q-testing leverages one LSTM along with its encoding module to extract features from state files. The states stored in memory buffer are also in the form of a feature vector. In this way, the time spent in comparing states will decrease since there is no need to extract features for all the remembered states. It is worth mentioning that other tasks that need state comparison or states compression may also benefit from such a module.

One of the challenges of applying the deep learning approach is that a large amount of training data is needed. We address this challenge in the following section.

4.3 Training Data Collection

In order to improve the model's performance, a wealth of samples should be collected. We collect and label the samples in 4 steps.

Instrumenting APKs. We collect plenty of commercial apps and each of them contains abundant scenarios. Since most of them

are closed-source apps, we conduct research on the Android framework and utilize Soot [40] to instrument several methods and statements of the APKs to collect runtime information which will provide more information to facilitate the later labeling process. Specifically, the instrumented methods and statements are corresponding to the start and destroy of Activity, Fragment or Dialog and often make a huge impact on the display of interface along with changes of scenarios. The invocations of these instrumented methods and statements usually mean a switch of scenarios. As for APKs do not support instrumenting, we will manually label samples collected from them.

Collecting Data. We implement a tool to automatically collect sample data. It randomly explores the applications and every time it executes an event, the states before and after the transition will be stored as a sample pair.

Augmenting Data. There are issues in the automatically collected data among which the most important one is that it only includes pairs where the two states can transit to each other with one event. This restriction will result in the different distribution between training data and actual data since a new state has to be compared with all the states with the same activity in memory.

In order to solve the problem, we make an augmentation of the collected data. More concretely, we divide the collected data into several groups *w.r.t.* their activities. Then new pairs are generated by randomly mapping the states in the same activity.

Labelling Data. After data collection steps, we look into the screenshots, the collected runtime information of every state pair and decide a label for the pair of whether it belongs to the same scenario or not.

The labeled samples are fed to the neural network for training. We pre-trained a model with 6058 pairs of samples for Q-testing. More details are in Section 5.

5 EVALUATION

In this section, we mainly inspect two parts: (1) the ability of the scenario division module; and (2) the ability of the whole testing tool to automatically covering codes and triggering bugs. We aim to answer the following research questions in our evaluation:

RQ1: ***Scenario Division.*** Is Q-testing able to determine different functional scenarios effectively?

RQ2: ***Code Coverage.*** How does Q-testing compare against state-of-the-art and state-of-practice testing tools with respect to code coverage?

RQ3: ***Fault Revelation.*** How does Q-testing compare against state-of-the-art and state-of-practice testing tools with respect to fault revelation?

5.1 Evaluation Setup

All the experiments are conducted on a physical machine with 4 cores 3.60GHz CPU and 16GB RAM on Ubuntu 16.04. Since the original Stoat [46] and Sapienz [33] are suggested to running on Android API level 19, We run all experiments on Android emulators which are configured with 2GB RAM, and the KitKat version (SDK 4.4, API level 19).

Scenario Division. We collect and label samples from 92 commercial Android apps with the approach described in Section 4.3. The automatic collecting tool explores each of the apps for at least one hour. Most of these popular apps comprise abundant scenarios that will benefit the classification accuracy of the neural network when applied to other applications. We conduct 5 rounds of 10-fold cross-validation. The division of data is conducted by apps that scenarios from the same application should not exist in both the training set and the testing set; otherwise, it may increase the classification accuracy but jeopardize the validation result.

Automated Testing. In this part of the evaluation, Q-testing is compared with Monkey [21], Stoat [46], and Sapienz [33], which are considered as the state-of-practice and state-of-the-art tools in Android GUI testing. Our benchmark consists of 50 open-source Android applications which are collected from two sources:

(1) Most of the applications are collected from related work [11, 41]. Apps selected by Choudhary et al. [10] have become a standard benchmark in evaluating automated testing tools and is adapted by several works [27, 33, 46]. Since many of these applications are quite out of date, we only involve those can be found in F-Droid [29] or GitHub [16]. We also exclude the projects that cannot be compiled due to long time no maintenance. After the filtering, 34 applications are selected.

(2) Over 40 percent of the applications selected above have less than 1k executable lines of codes (ELOC) which may be unable to reflect the difference between diversity exploration strategies. So we enrich the benchmark by adding 16 larger apps from the open-source apps list [39]. The applications are randomly selected from several commonly used categories (i.e., communication, news, education, tools) and most of them have more than 10k ELOC.

Additionally, applications that have been used to train scenario division module and gaming applications are excluded. Note that although the collected applications all have source-code, Q-testing can test application APKs without source-code. We use these open-source applications so that we can collect code coverage more accurately, analyze the root causes of faults with reference to the code, and better understand the testing behavior.

The testing time is set to one hour. We follow previous work [11, 24, 34] to set a 200 milliseconds wait interval for Monkey to partially avoid some abnormal behavior. Stoat's default time settings for 2 phases are one hour and two hours. We did not follow the allocation proportion as [46] in our one hour testing because we were not sure whether 20 minutes of modeling was enough. We conducted a small evaluation on several applications to compare the allocation of 40/20, 30/30, and 20/40 minutes for Stoat. The difference is not huge but the 30/30 minutes allocation does exceed. So all the apps are tested with this time allocation. As for Sapienz, we notice the evolution step can not be executed in some apps because of the large default setting for population size. So we run Sapienz with different population sizes for each app and pick the one with the highest code coverage. The selected setting is listed in Table 1. To mitigate randomness, for every testing tool, four test runs were conducted on the benchmark apps. We also enable Stoat and Sapienz to compute code coverage with Jacoco [14] for consistency.

5.2 Experimental Results

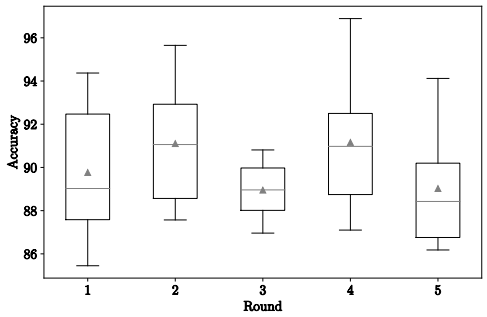


Figure 5: Accuracy of Scenario Division

RQ1: ***Scenario Division.*** We totally collect and label 6058 pairs of samples. For every round of 10-fold cross-validation, the samples are divided into 10 sets and every set is selected as the testing set by turns. The neural network is trained with approximately 90 percent of the data for 100 epochs which costs about half an hour on our experiment PC. After that, the trained model is applied to the testing set to complete a test. After 10 repeated operations, the average accuracy is calculated. Figure 5 shows the result of the 5 rounds of 10-fold cross-validation. The average accuracy is 89.80% and the lowest accuracy is above 85%, which indicates the model's ability to divide different scenarios.

We also undertake a careful analysis of the classification results. The examples in Figure 3, which are important scenarios in Android applications, are all correctly identified as the same scenario by our model. These are also samples difficult for previous criterion [6], including executable events and widget layout, to make a correct judgment.

As for those predicted mistakenly by our model, some of them are difficult to distinguish even by human users. Since the design of Android applications is quite flexible, it is sometimes hard to clearly define whether 2 states belong to the same scenario. Take Figure 6 as an example. The left state has a search bar on the top of several pieces of news which makes it look like a combination of searching scenario and browsing scenario. Even if the right state can be reached by simply scrolling down in the left state, it is hard to category the two states into the same scenario. Fortunately, our reinforcement learning framework is able to tolerate mistakes made by the scenario division module because of the bellman function. On the one hand, if the module makes a mistake and updates the event's value wrongly with a large number, its value will still be rectified later after several triggering and updating due to value propagation. On the other hand, if a new scenario which leads to important states is reached and the neural network mistakenly updates the event's value with a small number, the executed event's value will still be updated larger by the bellman function. With this function, all the subsequent events' value will make an effect on the previous one.

In conclusion, it is reliable to determine different scenarios and calculate reinforcement learning rewards with the scenario division module.

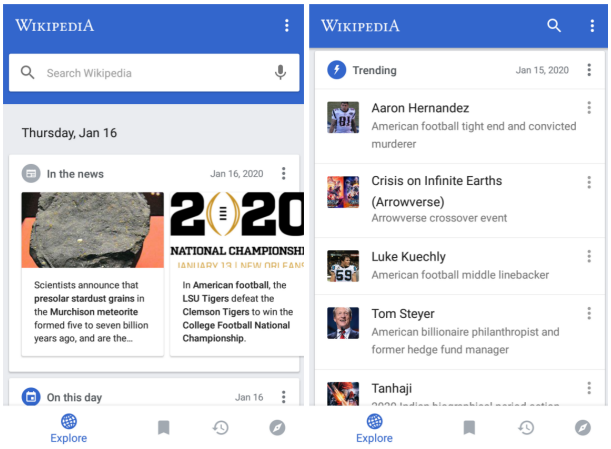


Figure 6: Example of Confusing Scenario

RQ2: ***Code Coverage.*** In this experiment, Q-testing uses the siamese network trained with the 6058 pairs of samples to divide scenarios.

Table 1 shows the average instruction coverage of the four test runs by Monkey, Stoat, Sapienz, and Q-testing on 50 open source applications. The apps are sorted by ELOC extracted from Jacoco coverage report and the first two apps are too small for Stoat to conduct the second phase. On average, Q-testing achieves 46.62% instruction coverage which is higher than Monkey (43.50%), Stoat (38.82%) and Sapienz (40.48%). Additionally, Q-testing achieves the highest code coverage for 33 of the open-source applications while the statistics for Monkey, Stoat, and Sapienz are 8, 7 and 9.

Coverage information is collected every 30 seconds during testing and Figure 7 depicts the progressive average code coverage for each tool in one hour. Monkey achieves the highest coverage within the first few minutes. That is mainly because it executes events at an extraordinarily high speed and lots of random events can lead it to new states in the very beginning. However, as the progress of testing continues, more and more redundant events are executed by Monkey and its efficiency starts to decrease. Q-testing takes the first place after about 5 minutes which confirms the effectiveness of our curiosity-driven strategy. The strategy to firstly discover different scenarios enables Q-testing to achieve high code coverage within a short time. This is extremely important when the testing budget is tight.

The efficiency of generating events is also important when testing time is limited. During one hour's testing, Monkey, Stoat, Sapienz, and Q-testing averagely generate approximately 53000, 1804, 29570 and 1693 events. As for Monkey, there is no need for it to analyze widget hierarchy before generating events. Besides, it can communicate directly with several essential Android services. These features enable it to generate events with high speed. Even if Monkey can generate much more events than others, most of them are redundant and may make no effect on the applications. Sapienz comes second in generating events as it is built upon Monkey. It is encouraging to see the number of executed events by Q-testing is close to that of Stoat's. Stoat can directly generate event sequences with reference to models, which benefits the testing efficiency. Q-testing does not

Table 1: Testing Results for Comparison

Subject		Coverage(%)				Faults				Setting
Name	ELOC	M	St	Sa	Q	M	St	Sa	Q	Sa
DivideAndConquer	80	91	-	95	95	0	-	0	0	50
QuickSettings	91	91	-	92	91	1	-	1	1	20
MunchLife	188	81	86	82	90	0	0	0	0	50
AnyCut	414	66	58	64	67	0	0	0	0	20
lockpatterngenerator	674	83	68	83	65	0	0	0	0	20
autoanswer	447	15	18	16	21	2	2	0	2	30
batterydog	468	59	51	60	49	0	0	1	1	50
Dumbphone	572	40	38	35	40	0	0	0	1	30
soundboard	641	38	44	32	46	3	0	0	1	30
whohasmystuff	744	77	75	68	78	1	1	0	2	50
zooborns	760	17	19	16	17	1	3	1	4	20
multismssender	862	55	55	59	45	3	1	0	0	50
alogcat	906	71	67	71	78	0	0	0	0	20
SmsScheduler	915	58	61	52	57	0	0	0	0	30
dialer	955	45	45	47	49	1	3	2	2	20
Talalarmo	1030	74	78	74	77	1	0	0	1	30
WeightChart	1054	67	58	67	78	6	2	3	2	50
alarmclock	1269	43	47	41	36	1	4	1	2	20
Notes	1910	59	55	54	73	3	0	3	2	20
PasswordMaker	1949	60	55	57	61	0	2	0	5	50
BudgetWatch	2143	42	45	47	61	1	0	1	6	40
manpages	2417	18	12	15	17	2	5	2	3	30
GoodWeather	2501	69	58	57	73	4	4	0	5	20
swiftp	3188	21	23	22	22	1	3	1	5	30
jamendo	3904	29	14	41	46	3	4	2	6	30
fillup	3967	58	57	43	60	1	5	0	1	40
sanity	4675	24	15	23	26	3	1	2	5	20
mileage	4711	44	35	45	42	2	8	2	7	20
importcontacts	4817	2	2	2	2	1	1	1	1	40
Tomdroid	4901	42	49	46	50	1	0	0	2	20
materialistic	6661	54	46	26	54	2	1	0	5	40
RadioBeacon	7459	35	39	32	41	4	4	1	11	20
TintBrowser	7575	38	40	16	43	0	4	0	0	30
AntennaPod	8153	43	34	24	42	9	2	0	9	30
keepassdroid	9035	11	8	11	8	0	0	0	2	20
ConnectBot	9090	21	23	16	26	2	1	0	5	50
APhotoManager	9751	50	36	46	55	3	1	6	5	50
BetterBatteryStats	10042	11	16	15	18	1	4	0	6	50
uhabits	10629	55	45	36	54	3	4	4	3	20
vanilla	10776	41	46	35	40	6	1	4	4	40
Timber	12004	43	30	31	36	9	6	4	10	40
AnyMemo	12414	31	20	44	46	6	15	2	10	30
Runnerup	16378	19	20	22	23	11	9	8	11	40
SuntimesWidget	16681	40	38	16	48	1	2	0	6	40
AmazeFileManager	17705	26	26	27	29	4	6	4	6	30
amme	21586	17	12	15	30	3	2	3	9	20
BookCatelogue	24378	28	30	29	35	2	1	0	6	50
Anki	28093	29	19	33	32	0	6	3	13	50
MyExpenses	29067	25	27	26	34	0	4	0	5	50
Signal	43954	21	20	20	26	1	8	0	4	30

have a model to guide the generation of test sequences. However, the Q-table acts like a lightweight model which maps explored states to actions and can save the time inferring executable events. What's more, the storage of states' vector rather than the original GUI hierarchy in memory set contributes to the improvement of efficiency too.

We also dissect the codes covered by different tools. Take GoodWeather which is a weather application as an example. In GoodWeather, a considerable portion of functions can be configured for variants via a 'Setting' option residing in a Sidebar. Q-testing outperforms other tools on GoodWeather mainly because it tries more settings during testing. Some of the settings have to be executed with at least 6 steps and it is a long event sequence for Android applications. There are a lot of candidate events in every step which

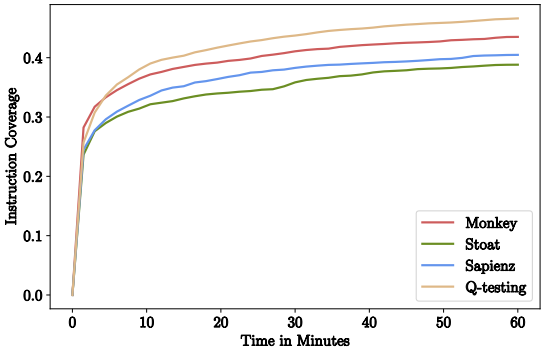


Figure 7: Progressive Instruction Coverage

makes the settings hard to trigger. To be more specific, we make an analysis of the testing behavior of Q-testing and discover that Q-testing tends to open the Sidebar (with 60.77% probability) in the main interface which is not viewed in Stoat (14.15%). Monkey and Sapienz do have the same tendency to open the Sidebar. That is because they would try to execute keyboard events and tend to click the button on the top left of the interface. However, keyboard events are not available in current mobiles, and clicking the top left area is now to open the Sidebar in GoodWeather. This phenomenon can also be viewed when the Sidebar is opened where Monkey would choose the first option with a greater possibility (43.20%) than the fourth 'Settings' option (13.02%). In Sidebar, where there are 6 candidate options and several other events (e.g. back, menu) to be chosen, Q-testing selects the 'Settings' option with a probability of 30.69%. This observed phenomenon indicates the power of curiosity-driven strategy and the value propagation property of Q-learning for exploring important scenarios which can lead to more new scenarios.

RQ3: ***Fault Revelation.*** Table 1 also shows the unique faults revealed by the four testing tools. We adopt the definition in Stoat where a fault is identified by crash or exception lines in stack traces (exceptions without the keywords of the AUT's package name are excluded). Q-testing detects the most faults for 27 of the experiment applications (tied first also counts) while Stoat detects the most for 12 of them and comes second. Q-testing reveals a total of 197 faults among the 50 applications, which is more effective than Monkey (109), Stoat (130), and Sapienz (62).

Figure 8 shows the pairwise comparison result of revealed faults between each 2 of the testing tools. The intersecting faults found by Stoat and Q-testing are 59 and most of them are triggered by system events. More than 30% of faults revealed by Sapienz is covered by Monkey since they share the same manner in injecting events. In addition, the numbers of unique faults, which are not revealed by the other 3 tools, for Q-testing, Monkey, Stoat, and Sapienz are 115, 63, 67, and 32. This shows the ability of Q-testing to trigger faults, which can not be replaced by other tools.

Most of the faults revealed by Q-testing have been reproduced and reported to the developers. So far, 22 reported issues have been confirmed to be first-time found real faults. We consider an issue confirmed if the developers respond in text clearly or add a 'Bug'

Table 2: Confirmed Issues Found by Q-testing

App Name	Exception	Description	Status	Issue URL
Anki-Android	ActivityNotFoundException	No Activity found to handle a View Intent	Fixed	github.com/ankidroid/Anki-Android/issues/5653
SuntimesWidget	NullPointerException	Multi thread related crash	Fixed	github.com/forrestguice/SuntimesWidget/issues/376
Runnerup	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Fixed	github.com/jonasoreland/runnerup/issues/862
Runnerup	IllegalStateException	Add a child view which already has a parent	Fixed	github.com/jonasoreland/runnerup/issues/863
Book-Catalogue	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Confirmed	github.com/eleybourn/Book-Catalogue/issues/835
manpages	InflateException	Error inflating class on binary XML file	Confirmed	github.com/Adonai/Man-Man/issues/19
manpages	NullPointerException	Crash after cache is cleared	Confirmed	github.com/Adonai/Man-Man/issues/20
Swiftp	NullPointerException	Unable to start FsWidgetProvider Receiver	Confirmed	github.com/ppareit/swiftp/issues/140
Swiftp	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Confirmed	github.com/ppareit/swiftp/issues/150
Budget-Watch	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Confirmed	github.com/brarcher/budget-watch/issues/202
Budget-Watch	WindowsLeakedException	TransactionActivity has leaked window that was originally added	Confirmed	github.com/brarcher/budget-watch/issues/203
Budget-Watch	NullPointerException	Unable to start ReceiptViewActivity	Confirmed	github.com/brarcher/budget-watch/issues/204
Notes	IndexOutOfBoundsException	Cursor out of bounds when handling Intent in NotificationService	Confirmed	github.com/SecUSo/privacy-friendly-notes/issues/77
Notes	IllegalArgumentException	Illegal argument in SketchActivity when editing reminder	Confirmed	github.com/SecUSo/privacy-friendly-notes/issues/78
AmazeFileManager	IllegalStateException	Illegal state while executing doInBackground()	Confirmed	github.com/TeamAmaze/AmazeFileManager/issues/1793
AmazeFileManager	WindowsLeakedException	Dialog's state is not preserved on screen orientation	Confirmed	github.com/TeamAmaze/AmazeFileManager/issues/1794
AmazeFileManager	ActivityNotFoundException	Intent cannot be handled when amaze cloud plugin is not available	Fixed	github.com/TeamAmaze/AmazeFileManager/issues/1795
AmazeFileManager	IndexOutOfBoundsException	Index out of bounds when cutting a folder and pasting within itself	Confirmed	github.com/TeamAmaze/AmazeFileManager/issues/1796
AmazeFileManager	NullPointerException	Unable to start TextEditorActivity	Fixed	github.com/TeamAmaze/AmazeFileManager/issues/1808
Uhabits	NullPointerException	WeekdayPickerDialog's not preserved results in null object reference	Fixed	github.com/iSoron/uhabits/issues/534
PasswordMaker	IndexOutOfBoundsException	Index out of bounds when deleting folder	Confirmed	github.com/passwordmaker/android-passwordmaker/issues/47
AnyMemo	NullPointerException	Crash when trying to paint because the library used is not maintained	Confirmed	github.com/helloworld1/AnyMemo/issues/488

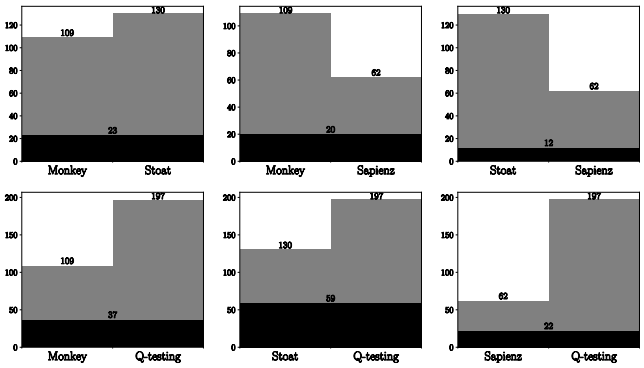


Figure 8: Pairwise Comparison on Fault Revelation

label manually. Among these 22 faults, 7 have been fixed. Table 2 list the confirmed issues along with the bug types, brief description, and statuses. Several bugs are quite concealable. For example, one multi-thread related crash in AmazeFileManager can only be triggered when a zip file, which is contained in another zip file, is continuously selected to be extracted. The event traces recorded by Q-testing help a lot in reproducing these faults. Considering that most of the apps have been maintained for quite a while and tested many times by previous work, revealing such an amount of new faults clearly demonstrates Q-testing's ability.

5.3 Threats to Validity

Internal Threats. The main internal threat lies in the choice of parameter settings for the four testing tools that may affect the testing results. In order to mitigate the threat, we try to choose their default settings as possible. For circumstances where default settings cannot be adopted, we conduct small-scale experiments and choose suitable settings before the formal evaluation. We cannot figure out a good setting for Sapienz's population size, so we run it with different settings and record the one that has the best testing effect.

The performance of siamese network is also an internal threat to validity. As for this threat, we conduct experiments to verify its ability. We will collect more training data and optimize the model in further work.

External Threats. The external threat mainly lies in the selection of subject apps and our results may not be generalized to other apps. We alleviate the threat by choosing subject apps from related work which include a standard benchmark. We further refine the benchmark by replacing out of date apps with large-scale ones from a popular open-source list.

6 RELATED WORK

Many techniques have been proposed to automate Android GUI testing. We broadly classify the technologies into four categories $w.r.t.$ their exploration strategies.

Random Testing. This kind of testing tools [21, 31] adopt random strategies to generate input for Android applications. Monkey [21], which is the most frequently used Android testing tools, generates pseudo-random streams of user events by randomly interacting with screen coordinates. This basic random strategy performs quite well on some benchmark apps [11]. However, the generated test cases contain a large number of noneffective or redundant events and this will be a threat to the testing effectiveness.

Some other tools [22, 42, 54] utilize fuzzing testing to generate intent inputs rather than explore applications' states to make the AUT crash or to reveal security issues. Q-testing also takes intent inputs into consideration when generating system-level events.

Model Based Testing. Model-based approaches [2, 3, 5, 6, 13, 23, 27, 44, 51–53] build models with dynamic or static strategies to describe the applications' behaviors and then derive test cases from the models to find bugs. Since the test cases are generated underlying the constructed model, its accuracy and completeness will be of great importance.

Stoat [46] utilizes a stochastic Finite State Machine model to describe the behavior of AUT. In the model construction process, it infers the executable events and prioritizes their executions according to factors like event type and executed frequency. Then in the test generation process, Stoat leverages MCMC sampling to direct the mutation of the model from which test cases are derived.

Q-testing leverages Q-table to record executable events in the covered state and the transition information is partially considered by the Bellman function. To some extent, Q-testing also benefits from model-based testing. However, Q-testing does not directly derive test cases from a model, which prevents it from the problem of inconsistent between model and applications' actual behavior. APE [24] makes some effort in model abstraction and refinement which alleviates the problem, but the model is still imprecise as internal states of variables, which will affect the applications' behavior, are still ignored due to limitation of the modeling language.

Systematic Testing. Symbolic execution and evolutionary algorithms are applied by systematic strategies [4, 15, 32] to generate specific input to cover hard-to-reach codes.

Sapienz [33] leverages a Pareto-optimal multi-objective search-based approach to maximize code coverage and bug revelation while at the same time minimizing the length of test sequences. It also reverses-engineering the APK to get statically-defined strings to generate specific input for text fields. Sapienz generates new test cases by random crossover and mutation, which will result in invalid sequences. The iterative evaluation of new generated test cases will also cost a lot of time.

Machine Learning Based Testing. Machine learning has been applied in Android GUI testing and related work can be divided into 2 categories. The first kind of approaches [7, 26, 28, 30] have an explicit training process to learn from the previous testing process and the learned experience will later be leveraged on new apps. QBE [26] learns from a set of Android applications the value of different types of events towards a particular objective like increasing activity coverage or detecting crashes. In order to make it possible for knowledge to be transferred between different applications, these approaches usually have to make abstractions on the apps' states. QBE divides states by the number of enabled actions where plenty of information is ignored and this may reduce testing effectiveness. Additionally, the design of Android applications is quite flexible, it may be useless to guide the testing of new applications with knowledge learned from others.

The other work tends to model independently for every app [1, 47, 48] or adapts a general model to the app under test [12]. Some of them [1, 47, 48] extend AutoBlackTest [35] and do not have an explicit training process. Similar to Q-testing, they also apply Q-learning to guide the exploration of Android applications. However, they simply rely on the difference between two states' executable events and their executing frequency to determine reward where the events' value can not be adjusted flexibly during testing. For example, an event connecting two states with quite different events will always be encouraged to execute when most events are executed many times. Q-testing tackles this problem by using memory and scenario division module to steer the testing towards unfamiliar functionalities with high efficiency.

Reinforcement learning is also applied to other testing work. For example, Wuji [55] combines reinforcement learning with evolutionary algorithms to test games. Retecs [45] uses RL to guide test prioritization and selection in regression testing.

7 CONCLUSION

In this paper, we propose Q-testing, a Q-learning based approach for Android automated testing. Q-testing leverages Q-table as a lightweight model while exploring unfamiliar functionalities with a curiosity-driven strategy. To effectively determine the reward for Q-learning and to further guide the exploration, a scenario division module which distinguishes functional scenarios using a neural network is proposed. Experiments show that Q-testing outperforms the state-of-the-art and state-of-practice Android GUI testing tools in both code covering and fault detection.

ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program (Grant No. 2017YFB1001801), the National Natural Science Foundation (Nos. 61632015, 61972193), and the Fundamental Research Funds for the Central Universities (Nos. 14380022, 14380020) of China.

REFERENCES

[1] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. Reinforcement learning for Android GUI testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 2–8.

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

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

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

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

[6] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *2016 Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 238–249.

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

[8] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a "siamese" time delay neural network. In *Advances in neural information processing systems*. 737–744.

[9] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A Efros. 2019. Large-scale study of curiosity-driven learning. In *7th International Conference on Learning Representations (ICLR)*.

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

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

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

[13] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd*

IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007. ACM, 31–36.
[14] EclEmma. 2020. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/index.html
[15] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *2018 Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 419–429.
[16] GitHub. 2020. GitHub. https://github.com/
[17] Google. 2019. Broadcasts overview. https://developer.android.google.cn/guide/components/broadcasts
[18] Google. 2019. ListView. https://developer.android.google.cn/reference/android/widget/ListView
[19] Google. 2019. Recyclerview. https://developer.android.google.cn/reference/androidx/recyclerview/widget/RecyclerView
[20] Google. 2019. UI Automator. https://developer.android.com/training/testing/ui-automator
[21] Google. 2019. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey
[22] NCC group. 2012. Intent Fuzzer. https://www.nccgroup.trust/us/our-research/intent-fuzzer/
[23] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. 2017. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 103–114.
[24] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 269–280.
[25] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. 2016. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*. 1109–1117.
[26] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 105–115.
[27] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for Android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 115–127.
[28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. 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)*. IEEE, 1070–1073.
[29] F-Droid Limited. 2020. F-Droid - Free and Open Source Android App Repository. https://f-droid.org/
[30] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.
[31] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.
[32] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.
[33] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.
[34] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 16–26.
[35] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. 2012. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 81–90.
[36] Jonas Mueller and Aditya Thyagarajan. 2016. Siamese Recurrent Architectures for Learning Sentence Similarity. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 2786–2792.
[37] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. 2016. Learning text similarity with siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*. 148–157.
[38] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. 2017. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*. 2778–2787.
[39] pcqpcq. 2020. Open-Source Android Apps. https://github.com/pcqpcq/open-source-android-apps
[40] Sable. 2019. Soot - A framework for analyzing and transforming Java and Android applications. https://sable.github.io/soot/
[41] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. Patdroid: permission-aware gui testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 220–232.
[42] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. ACM, 1–5.
[43] Nikolay Savinov, Anton Raichuk, Raphaël Marinier, Damien Vincent, Marc Pollefeys, Timothy Lillicrap, and Sylvain Gelly. 2019. Episodic curiosity through reachability. In *7th International Conference on Learning Representations (ICLR)*.
[44] M Shafique and Y Labiche. 2010. A systematic review of model based testing tool support, Technical Report SCE-10-04. *Carleton University, Canada* (2010).
[45] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 12–22.
[46] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256.
[47] Tuyet Vuong and Shingo Takada. 2019. Semantic analysis for deep Q-network in android GUI testing. In *31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*. Knowledge Systems Institute Graduate School, 123–128.
[48] Thi Anh Tuyet Vuong and Shingo Takada. 2018. A reinforcement learning based approach to automated testing of Android applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 31–37.
[49] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.
[50] Yoram Wurmser. 2018. Mobile Time Spent 2018. https://www.emarketer.com/content/mobile-time-spent-2018
[51] Jiwei Yan, Linjie Pan, Yaqi Li, Jun Yan, and Jian Zhang. 2018. LAND: a user-friendly and customizable test generation tool for Android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 360–363.
[52] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. 2017. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 42–53.
[53] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.
[54] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM, 68.
[55] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yinfeng Chen, and Changjie Fan. 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 772–784.