Effective, Platform-Independent GUI Testing via Image Embedding and Reinforcement Learning

SHENGCHENG YU, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHUNRONG FANG∗, State Key Laboratory for Novel Software Technology, Nanjing University, China
XIN LI, State Key Laboratory for Novel Software Technology, Nanjing University, China
YUCHEN LING, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENDONG SU, Department of Computer Science, ETH Zurich, Switzerland

Software applications (apps) have been playing an increasingly important role in various aspects of society. In particular, mobile apps and web apps are the most prevalent among all applications and are widely used in various industries as well as in people’s daily lives. To help ensure mobile and web app quality, many approaches have been introduced to improve app GUI testing via automated exploration, including random testing, model-based testing, learning-based testing, etc. Despite the extensive effort, existing approaches are still limited in reaching high code coverage, constructing high-quality models, and being generally applicable. Reinforcement learning-based approaches, as a group of representative and advanced approaches for automated GUI exploration testing, are faced with difficult challenges, including effective app state abstraction, reward function design, etc. Moreover, they heavily depend on the specific execution platforms (i.e., Android or Web), thus leading to poor generalizability and being unable to adapt to different platforms.

This work specifically tackles these challenges based on the high-level observation that apps from distinct platforms share commonalities in GUI design. Indeed, we propose PIRLTest, an effective platform-independent approach for app testing. Specifically, PIRLTest utilizes computer vision and reinforcement learning techniques in a novel, synergistic manner for automated testing. It extracts the GUI widgets from GUI pages and characterizes the corresponding GUI layouts, embedding the GUI pages as states. The app GUI state combines the macroscopic perspective (app GUI layout) and the microscopic perspective (app GUI widget), and attaches the critical semantic information from GUI images. This enables PIRLTest to be platform-independent and makes the testing approach generally applicable on different platforms. PIRLTest explores apps with the guidance of a curiosity-driven strategy, which uses a Q-network to estimate the values of specific state-action pairs to encourage more exploration in uncovered pages without platform dependency. The exploration will be assigned with rewards for all actions, which are designed considering both the app GUI states and the concrete widgets, to help the framework explore more uncovered pages. We conduct an empirical study on 20 mobile apps and 5 web apps, and the results show that PIRLTest is zero-cost when being adapted to different platforms, and can perform better than the baselines, covering

∗Chunrong Fang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s).
ACM 1557-7392/2024/6-ART
https://doi.org/10.1145/3674728

ACM Trans. Softw. Eng. Methodol.
6.3–41.4% more code on mobile apps and 1.5–51.1% more code on web apps. PIRLTest is capable of detecting 128 unique bugs on mobile and web apps, including 100 bugs that cannot be detected by the baselines.

**CCS Concepts:** • Software and its engineering → Software testing and debugging.

Additional Key Words and Phrases: Software Testing, Platform-Independent Testing, Reinforcement Learning, GUI Image Understanding

## 1 INTRODUCTION

Testing is an effective method to ensure software quality. Current approaches utilize different strategies to automatically test the applications (app)\(^1\)\[^1\], but such approaches still leave quite room for improvement. Manually written test scripts based on specific frameworks\(^4\)\[^4\],\[^60\] are commonly used in app testing\[^27\],\[^32\],\[^33\],\[^73\]. Testers are required to identify the target widgets and assign the test events. However, the manual work is quite a waste of time and human resources, creating a great burden on app developers. The involvement of manual effort of testers with different expertise makes test scripts’ quality ranges widely.

One most widely used automated testing strategy is the random-based exploration\[^28\],\[^45\], which can generate pseudo-random test events to fuzz the apps with relatively high efficiency brought by the random feature. However, most of the generated test events are invalid, which may not trigger interactive responses. Monkey\[^28\], as a representative tool, may generate test events like clicking on blank areas, inputting texts into ImageView widgets, etc. To increase the exploration effectiveness, model-based strategy\[^6\],\[^11\],\[^46\],\[^47\],\[^62\] is adopted to construct models for apps under test with dynamic or static app analysis technologies. Two main factors, the constructed model itself and the exploration strategy, determine the effectiveness of the model-based approaches. Due to the limitations of existing app analysis technologies, the generated models cannot reach high code coverage, thus imposing restrictions on the model-based strategy. Besides, specific app states require domain knowledge to generate valid inputs, which is tough for existing exploration strategies of automated testing. Learning-based technologies have increasing adaptive applications in automated testing. Learning-based Reinforcement learning (RL)\[^35\],\[^65\] is an effective algorithm to help effectively explore the app states. Table 1 shows the comparison of representative RL-based approaches and PIRLTest. The “Platform” column show the platform supporting situation. The “State Abstraction” column shows the basis for state abstraction. The “Supported Action Type” column shows how the explored states and corresponding reward values are stored and utilized for further exploration guidance. The “Reward Function Basis” column shows the important factors considered in the reward function design.

However, it is still a tricky problem to effectively ensure the app quality due to its large exploration space. Even if it is shown in many studies that the RL framework is an advanced solution to app testing, current RL-based approaches are still faced with many problems. Existing approaches rely heavily on the platform features or interfaces, which prevents them from being adapted across different platforms with an effective abstraction to the app state. For example, Q-testing\[^52\] utilizes the layout files of Android apps as the app states; WebExplor\[^76\] utilizes the front page HTML code as the app states. This brings much extra overhead when developers conduct testing on different platforms or they migrate tests among different platforms. Moreover, app state abstraction is an important basis for the reward function design, which assigns different reward values to actions that make covered or uncovered pages. However, according to our investigation and existing work\[^18\],\[^73\], we find that apps of different platforms adopt highly similar widget styles\(^2\)\[^2\],\[^e.g., Button, TextField, ImageView,\] etc.). This suggests that the GUI images can be utilized as app states to guide the RL-based strategy to make the exploration independently adaptive to apps of different platforms. Another problem of app state abstraction

---

\(^1\)“Software” and “application” are used interchangeably in this paper.

\(^2\)Customized rendering is not considered.
Table 1. Comparison among Representative RL-based Approaches. Platform: support for mobile or web platforms. State Abstraction: support for mobile or web platforms. Supported Action Type: supported action numbers of different types on mobile and web platforms. Exploration Strategy: Storage and utilization of explored states and corresponding reward values for further exploration guidance. Reward Function Basis: factors considered in the reward function design (closely related to State Abstraction).

| Approach      | Platform | State Abstraction | Supported Action Type (mobile/web) | Exploration Strategy | Reward Function Basis          |
|---------------|----------|-------------------|-----------------------------------|----------------------|-------------------------------|
| QBE [36]      | ✓        | Layout File       | 5/0 1/0 5/0                       | Q-table              | Not Specified                 |
| Q-testing [52]| ✓        | Layout File       | 4/0 2/0 3/0                       | Q-table              | Activity Comparison           |
| WebExplor [76]| ✓ ✓      | HTML Doc          | 0/1 0/0 0/0                       | Q-function           | State-Action-State-Tuple      |
| ARES [58]     | ✓        | Layout File       | 4/0 0/0 2/0                       | Neural Network       | Action Comparison             |
| Fastbot2 [44] | ✓        | Layout File       | 2/0 0/0 0/0                       | Q-table              | Action Comparison             |
| DinoDroid [75]| ✓        | Event Flow        | 4/0 1/0 2/0                       | DQN                  | Code Coverage & Bug Detection |
| PIRLTest      | ✓ ✓      | GUI Widget & GUI Layout | 5|6 3|3 6|5 | DQN                       | GUI State Transition Times & GUI Widget Exploration Rate |

of current automated testing approaches is widget identification. Current approaches rely on the layout files to identify the target widgets [52]. However, most apps adopt a hybrid version and embed Canvas elements or customized elements into the app pages. Such GUI widgets cannot be identified with existing approaches. As is often the case, a group of widgets will be considered as only one single widget, which greatly affects the exploration effectiveness. Consequently, identifying widgets directly from the app GUI is a good choice because it views the app GUI just like app users.

In order to dispose of the aforementioned challenges, we propose a novel approach, named PIRLTest, to realize Platform-Independent GUI Testing via image embedding and Reinforcement Learning. We adopt a deep Q-network (DQN) [49] to guide PIRLTest to explore more app states. First, we propose a novel algorithm to characterize the app state from the perspective of GUI to avoid dependency on platform features. The accuracy of the state characterization of an app can significantly affect the exploration capabilities of the RL framework. Specifically, we combine a microscopic perspective (i.e., widget feature) and a macroscopic perspective (i.e., structure feature) to complete the app GUI embedding. For the widget feature, we extract all the widget images and their coordinates with the computer vision (CV) algorithms. The embedded widget feature is composed of three parts. First, the widget image is embedded in a vector with a convolutional neural network (CNN); second, the widget coordinate is fed into a CNN to generate a vector after the corresponding pixels of the widget in the app page to black and other pixels to white; the third part of the widget feature is a 14-length one-hot vector representing the widget type recognized by a CNN model. Then, the three vectors are concatenated to represent a widget, and we get the average vector of all widget features. For the structure feature, we use a tree structure to represent the layout relationships of all the widgets, neglecting the concrete widget images. The trees are transformed into strings with curly braces to keep the relative relationships. The strings are fed into a recurrent neural network (RNN) to obtain a vector. The widget feature and the structure feature are concatenated to accomplish the app state (i.e., page) embedding.

PIRLTest adopts the curiosity-driven [12] strategy in the DQN algorithm to explore more app pages with the test event (i.e., action) execution. Specifically, given the app state, executable actions are inferred, then the Q-network of PIRLTest, trained with the history exploration data stored in the exploration memory, is used to determine the values of all state-action pairs. The values of state-action pairs are transferred to the execution probabilities based on the Boltzmann strategy [14] to avoid the “exploration-exploitation dilemma” [57], and the state-action pair with the highest probability is executed. The reward function is a critical part of the RL framework to train the Q-network to better guide the algorithm to explore more uncovered pages. The design of

ACM Trans. Softw. Eng. Methodol.
the reward function in PIRLTest is based on the exploration rate of one specific app page and the page transition times. Such a design relies heavily on the app GUI state representation with a combination of the macroscopic perspective (app GUI layout) and the microscopic perspective (app GUI widget). Exploration rate refers to the explored action percentage in specific app pages, and page transition times can tell how many times this page is transited to. Both are based on the page similarity with the app GUI image embedding.

We conduct an empirical study to evaluate PIRLTest on different platforms. The experiment results show that PIRLTest achieves satisfying effectiveness on different platforms, with no extra modification or changes to generate tests on different platforms. PIRLTest covers 6.3–41.4% more code on mobile apps and 1.5–51.1% more code on web apps, and it covers a large percentage of code that is not covered by the baselines. PIRLTest is capable of detecting 128 unique bugs, including 100 ones that cannot be detected by the baselines.

We declare the following noteworthy contributions of this paper:

• We introduce a platform-independent GUI testing approach via image embedding and reinforcement learning, which is zero-cost when being adapted to different platforms.
• We propose a novel algorithm to abstract the app GUI states with the GUI image embedding, including the widget extraction (microscopic perspective) and the layout characterization (macroscopic perspective). This algorithm helps effectively characterize the States in the RL model.
• We propose a novel reward function design in the RL framework, with a comprehensive consideration of the exploration of both app GUI as a whole and concrete GUI widgets.
• We implement a tool and conduct an empirical evaluation of the effectiveness of PIRLTest on both mobile apps and web apps, which shows the outstanding performance of PIRLTest over the representative baselines.

More information and the reproduction package is available on https://sites.google.com/view/pirltest.

2 BACKGROUND & MOTIVATION

In this section, we present the background and the motivation of this work, including the preliminaries of reinforcement learning, and the limitations of current approaches. We present the commonalities of mobile apps and web apps as the intuition to complete this paper.

2.1 Deep Reinforcement Learning

Reinforcement learning is a group of machine learning algorithms that make decisions and change behaviors based on environmental feedback to maximize expected benefits [35]. Reinforcement learning can be modeled as a Markov decision process (MDP) as shown in Fig. 1. The MDP can be formalized as a four-tuple \( (S, A, R, P) \), where \( S \) refers to the set of all the states, \( A \) refers to the set of all the actions, \( R \) refers to the reward function: \( S \times A \rightarrow R \), and \( P \) refers to the transition probability function: \( P (s, a, s') = P (s_{t+1} = s' \mid s_t = s, a_t = a) \). In the scenario of automated testing, state means the current situation of the app under test, like the widgets and their contents presented on the app GUI, and the corresponding GUI layout, i.e., relationships among different GUI widgets. Action means the events that can lead to the change of app GUI, like the activity transition. Generally speaking, at each timestamp \( t \), the agent senses the state \( s_t \) from the environment and then selects an action \( a_t \) based on a specific policy \( \pi (a_t | s_t) \). As the \( a_t \) is executed, a state transition to a new state \( s_{t+1} \) will happen, and a reward \( r_t \) should be assigned. The object of RL is to train such an agent to maximize the expected discounted cumulative reward.

When traditional tabular RL algorithms are faced with the high-dimension or unbounded state and action spaces, it is impractical and unstable to learn an efficient policy merely based on such tabular algorithms. Therefore, a group of more robust RL algorithms combining the deep neural networks, named deep RL, is proposed to overcome the shortcomings. Deep Q-network [49] is one model-free representative. In DQN, a neural network is
used to approximate the state-action value function $Q^\pi$. The neural network is trained to estimate the expected discounted cumulative reward under the optimal policy $\pi^*$. With the combination of reinforcement learning and deep learning, i.e., the DQN, it will become more effective to explore the environment.

2.2 Commonalities of Mobile & Web Apps

In order to apply the PIRLTest on different platforms, including web platform and mobile platform, one significant prerequisite is the commonalities in app GUI of mobile apps and web apps. The intuition of PIRLTest is to extract all the widgets from the app pages for the exploration of the RL algorithm (details in Section 3.1). The app pages are composed of widgets. Besides, the widgets used in mobile apps$^3$ and in web apps$^4$ can be mapped. We list several widely used widgets and their mapping relationships. As is depicted in Table 2, the Android widget classes correspond to specific HTML tags. From the visual aspect, the widget pairs share universal appearances.

| Android Class | Web (HTML) Tag |
|---------------|----------------|
| Button        | <button>       |
| TextView      | <p>            |
| EditText      | <input>        |
| CheckBox      | <input type="checkbox"> |
| ImageButton   | <input type="image"> |
| ImageView     | <img>          |
| RadioButton   | <input type="radio"> |
| Switch        | <input type="checkbox"> |
| SeekBar       | <input type="range"> |
| ProgressBar   | <progress>     |

Moreover, corresponding widgets share similar applicable actions (details in Section 3.2). For example, for a clickable Button in a mobile or a web app, the possible actions are click, right-click, etc.; for an editable TextField (or <input> tag), the possible actions are input, long-click, etc.

$^3$https://developer.android.com/reference/android/view/View

$^4$https://www.w3.org/
Considering the above commonalities between widgets of mobile apps and web apps, and starting both from the visual aspect and the functionality correspondence, we can develop an effective approach that utilizes the GUI information with the help of CV technologies to guide the exploration of the reinforcement learning algorithm.

The commonalities of apps on different platforms inspire us to develop the platform-independent automated testing approach. First, from the user perspective, although apps on different platforms may have differences in their implementation, they tend to have similar GUI designs and GUI layouts [73]. It is nature for app users to perceive the apps with a platform-independent cognition. Apps on different platforms also adopt similar widget types and interactions from the user perspective. The RL algorithm is designed to simulate the testing process of app users to explore different activities in the apps. Therefore, it is reasonable to design the platform-independent automated testing approach from the actual user perspective. Second, from the app developer perspective, they always need to develop different versions on different platforms to attract more users with different devices. It forms a bar for app developers to use different tools for the automated testing for different platforms. As far as we know, some commercial companies hope to use a universal tool on all platforms instead of individual tools on different platforms, but none of existing tools can support this. In other fields of automated testing, researchers are also developing platform-independent tools, like the Appium\(^5\), one of the most popular tools to assist develop test scripts on different platforms, with which testers still need to maintain scripts with the same structure but different concrete contents (i.e., widget identifiers). LIRAT by Yu et al. [73] is a fully platform-independent GUI test script record and replay tool that uses the visual-based app GUI widget matching. Therefore, we believe our work is significant as it is the first platform-independent exploration testing work that can be applied on different platforms.

2.3 Limitations of Current Approaches

For current approaches that apply RL algorithms in app testing, like [52, 58, 76, 77], almost all of them depend on specific tools to capture the GUI structures. For example, UIAutomator\(^6\) is a tool that can extract the mobile app page elements and the structure. WebExplor obtains the HTML elements from the browser. However, one prominent problem is that more and more pages adopt a hybrid version or have a highly customized widget due to the deeper customization of app GUIs. Under such circumstances, these widgets are hardly possible to be obtained.

---

\(^5\)https://appium.io/
\(^6\)https://developer.android.com/training/testing/other-components/ui-automator
Fig. 2(a) shows an example of a mobile app. In this Rule app, the rule is a customized widget. The blue-edge widget can be dragged, and the gear-like button can be clicked. However, owing that this widget is a customized one, the rule is recognized as a whole widget in the class android.view.View. Also, the blue-edge widget and the gear-like button cannot be obtained by program analysis tool because they are just images shown on the Canvas widget, which may negatively affect the app exploration effectiveness.

Fig. 2(b) gives an example from the web version of Google Maps. The red-circled texts are clickable and will trigger an information page, showing the details about the location. However, texts of location names are not accessible from the HTML layout structure files given that the whole page is a simple Canvas widget from the perspective of the HTML files.

For PIRLTest, which solves the limitations by utilizing the computer vision technologies, the widgets mentioned above can be identified without relying on the underlying layout files. More details are introduced in Section 3.1.1 and Section 3.1.2.

3 APPROACH

In this section, we present the detailed design of PIRLTest. The app GUI screenshots are processed with CV algorithms to embed the app page into a vector so as to represent the states (Fig. 3 ❶), combining the macroscopic perspective (app GUI layout) and the microscopic perspective (app GUI widget). Based on the extracted widgets during the app GUI processing, candidate actions are generated and assigned to the corresponding widgets (Fig. 3 ❷), then PIRLTest embed the actions into vectors. Afterwards, PIRLTest calculates the state-action values with the Q-network and assigns different probabilities to the actions according to the values (Fig. 3 ❸). The Q-network is trained with the data from explored states, which are stored in the memory, as the curiosity-driven strategy will encourage PIRLTest to explore different states based on the reward calculation (Fig. 3 ❹). In the following sections, we will give a detailed depiction of the PIRLTest design.

3.1 App GUI Processing

One of the most significant parts of an RL framework is to abstract the state, which is a prerequisite for the reward calculation to reward the new state exploration. Traditional learning-based approaches tend to dump the XML layout files to abstract app state, which is quite inaccurate according to the challenges mentioned in Section 2.3. Therefore, we start the app state embedding directly from the GUI images. App GUI (activity) is composed of different types of GUI widgets, which are the basic elements. The layout of such widgets, i.e., the arrangements, describes the basic skeleton of the app GUI. Therefore, we focus on the GUI widgets and the widget layout during
the state embedding. The widget layout describes the structure of the whole app GUI from the macroscopic perspective, and the widgets themselves describe the concrete contents of the app GUI from the microscopic perspective. Combining the two perspectives can help more precisely embed the app GUI images. The following two sections describe the detailed processing of widget extraction & embedding and layout characterization & embedding.

3.1.1 GUI Widget Extraction & Embedding. The first step of GUI processing is to extract GUI widgets from the app screenshots, widgets are the basic components of app GUI images and are also the prerequisite for layout characterization. We adopt the classic computer vision algorithm, the Canny [13] edge detection, which is widely used in widget recognition in GUI analysis [18], to our approach. Targeting at one specific app screenshot of being turned grayscale, the Canny algorithm identifies the edges of widgets. Then the edges will be extracted to form an image of binary thresholding, where edges are represented with white pixels and non-edges with black pixels. Afterward, the edges of the widgets are obtained. However, there are still some noise data to be processed, e.g., the misrecognized subtle areas. Such noise data will not be assigned with interactable attributes, so we have to eliminate such areas to avoid misleading during the RL process. The accordance is the size ratio of the widget to the whole screenshot, and we set a threshold, following existing studies [18, 73], to eliminate the subtle misrecognized areas.

We use the IoU metric to evaluate the effectiveness, which is a widely used metric [18] for object detection and means the overlap between the detected widget areas and the labeled ground-truth widget areas. We set the widgets with IoU bigger than 0.8 as true positive (the threshold is set according to existing studies [18, 73]). The larger IoU is, the more accurate the detected results are. In the evaluation, true positive refers to a detected widget which matches a ground truth widget. False positive refers to a detected widget which does not match any ground truth widgets. False negative refers to a ground truth widget which is not matched by any detected boxes. We first construct an evaluation dataset. We invite experts to label and cross-validate 17934 widgets on 1000 GUI screenshots randomly collected from different categories of apps (no overlap with our experiment subjects). Then, we run the Canny algorithm on the GUI screenshots to evaluate the effectiveness. The precision value is 94.73%, the recall value is 99.09%, the F-measure value is 96.86%, and the accuracy is 93.92%. The results reflect the high effectiveness of the Canny algorithm in our scenario in detecting GUI widgets from the visual perspective.

All obtained single widgets view the app GUI from a microscopic perspective. We consider three attributes of a widget image for each widget in the app GUI: the widget image, the widget location, and the widget type. These three attributes are combined to determine two widgets on different app GUI images are the same or similar widgets.

**Widget Image** refers to the widget extracted from the GUI screenshot. For each widget, we feed the widget image into a convolutional neural network to embed it into a 4096-dimension vector. Specifically, the CNN we use here is a VGG-16 model [61]. The model is a pre-trained model in [72] with the purpose of classifying widget types. The model is trained with a dataset containing over 36k widget images manually collected, labeled, and verified by the authors. In order to obtain the embedding vector, we delete the last layer of the model, which outputs the probabilities indicating the types. The last but one layer is a FullyConnected layer containing 4096 neurons, and it will output a 4096-dimension vector, which can be viewed to represent the features of the widget images [34, 40].

**Widget Location** indicates the location of the widget within the page. Traditionally, in order to represent the coordinate of a widget, common practices use four integers: the coordinate of the left-upper corner and the width and height of the widget. The coordinates can be obtained during the widget extraction. However, if we directly use the four integers to embed location information, the weight of location to the final widget feature is too low and will be neglected by the model. Therefore, we use a black-and-white GUI image to represent the widget location. We turn the widget part of a GUI image into all-white and the other part into all-black. Then, the
A transformed GUI image is fed into the VGG-16 model mentioned above to obtain a 4096-dimension vector to represent the location features, which is the same practice on widget images.

**Widget Type** is identified by a convolutional neural network. Widget type has a close relationship to the applicable actions, and the type attribute will affect the exploration. Thus, we consider embedding the widget type. As mentioned in the **Widget Image** part, the VGG-16 model is trained with the purpose of identifying the widget type. The model covers 14 widely used widgets [72] which are applicable to different platforms. We use a 14-dimension one-hot vector to represent the widget type. Compared with the widget image and the widget location, the importance of the widget type is weaker, so we do not expand the dimensions. We evaluate the classification results on 14 categories with precision, recall, F-measure, and accuracy. The results are 90.05%, 89.97%, 90.01%, and 89.98%, respectively. The results show the good performance of the classification results.

The three kinds of widget embedding results are concatenated to form the widget representation vector, with a length of 8206 dimensions ($2 \times 4096 + 14$). The 8206-dimension vector is the embedding representation of one single widget on the app GUI.

3.1.2 **GUI Layout Characterization & Embedding.** After obtaining the widgets from the app screenshots, the layout of the app screenshots should be characterized, which illustrates the overall structure of the screenshot and the relative positions among all widgets. Beyond concrete widgets, layout characterization effectively reflects the whole panorama of the GUI structure, which is important in state determination. During the GUI design, the designers tend to arrange the widgets with close semantics together, and arrange widgets with different ones apart [38, 56, 59]. Consequently, we characterize the screenshot layout with a four-level hierarchy (Fig. 4), the whole app GUI, the Group, the Line, and the Column [73]. Group and Line are horizontal operations, and Column is the vertical operation. Group is a rough characterization (red contours in Fig. 4) and the design intuition is that some widgets should be relatively close to each other. Widgets within the same group may occupy different Lines due to their sizes. We further divide the widgets within each Group into different Lines (blue contours in Fig. 4) and different Columns (green contours in Fig. 4) according to the coordinates. Obtaining Columns in different Lines are conducted respectively, so different Lines may include different amounts of Columns.

![Image of GUI layout characterization](image)

**Fig. 4. Layout Characterization of App GUI**

The GUI layout views the app GUI from a macroscopic perspective. As described in Section 3.1.2, we characterize the layout with a hierarchy structure. The hierarchy structure can be easily transformed into a tree structure as
shown in Fig. 5. The left part of Fig. 5 shows a simplified example of layout characterization, where red lines identify the Group, blue lines identify the Line, and the green lines identify the Column. In the right part of Fig. 5, the first level represents the whole app screenshot, the second level represents the Groups, the third level represents the Lines, and the fourth level represents the Columns, which are actually the real widgets in the app screenshot. Due to the special design of the layout characterization, the transformed tree structure will have four levels.

To further make use of the layout tree, we transform it into a string. One common practice is to use curly braces to indicate the level relationship. The elements within the curly braces refer to the children nodes of the element before the curly brace. Strings with curly braces can be one-to-one mapped to specific GUI trees.

Then, we embed the GUI layout with the LSTM model [29]. LSTM is a suitable option because it can effectively embed our input strings indicating the GUI layout structure, which is not quite complex, and it will not bring much extra overhead when processing the GUI layout. LSTM helps achieve a balance between effectiveness and efficiency. In order to train the LSTM model, we randomly generate 100 different GUI layout strings, and the dataset size is 10000 (100 × 100) pairs. The dataset is divided into the training set, validation set, and test set at the ratio of 7:1:2, following the common practice. The purpose of the LSTM is to fit the tree edit distance [53, 54]. The output is a 512-dimension vector after a FullyConnected layer, and the generated vector represents the embedded GUI layout.

In order to study whether there might be potential impact of the random generated strings, we repeat the process for 30 times, following the common practice of software engineering research [5]. Since the target of the layout feature extraction model is to fit the tree edit distance, we calculate the sum value of differences between actual distances and predicted distances in each turn. We conduct the t-test on the results of the 30 repetitions, and the null hypothesis is that there is significant difference among different turns. The significance value p is calculated as 0.00018, which is less than 0.05. The null hypothesis is rejected, which means there is no significant difference in the results of different turns.

### 3.1.3 State Representation

In order to comprehensively represent the app current state, we combine the microscopic perspective and the macroscopic perspective, i.e., the widget embedding and the layout embedding. The widget embedding results and the layout embedding results are merged to obtain the final state (app page) representation. For each widget in the app screenshot, we obtain an 8206-dimension vector. However, the widget amount of different GUI screenshots are different, so if we directly concatenate the vectors, we may get variable-length vectors as widget numbers are different, which do not fit the requirements of the input of the RL framework.
Therefore, we conduct an average calculation of all the widget embedding vectors and obtain one single 8206-dimension vector that represents all the widgets. Then, we concatenate the widget embedding vector with the layout embedding vector to obtain the final app page vector and view it as a corresponding state of the page in the RL framework. The length of the final vector representing the app GUI is 8718 ($2 \times 4096 + 14 + 512$). In this work, we propose a novel app GUI state abstraction algorithm with the GUI image embedding. Our algorithm is designed to simulate the perspective of real app users to view the app GUI, including the widget extraction (microscopic perspective) and the layout characterization (macroscopic perspective). Our work is the first one to combine different granularities of GUI features into GUI state abstraction, and the results show that the abstraction is effective. Compared with existing learning-based automated testing methods, which directly extract widgets from layout files as the app states, PIRLTest can better help better characterize the app states from different perspectives, and can more realistically reflect the actual GUI widgets on the app GUI, because some widgets are hidden in other widgets from the layout files (HTML), like the Canvas widgets.

3.2 Action Generation

Actions are another significant part of the whole reinforcement learning framework. In the automated software testing scenario, the action is limited. As shown in Table 3, we list all the actions that are considered in PIRLTest. In software testing, the actions can be divided into three main categories: the widget actions, the page actions, and the system actions. Different actions may have different targets and parameters. In this paper, we consider in total 17 different actions. Some actions apply to both mobile and web platforms, and some are unique to a specific platform.

| Platform | Widget Actions | Page Actions | System Actions |
|----------|----------------|--------------|----------------|
| Both     | click, input, drag, double click | swipe, split screen | return, back switch, access grant, access deny, network switch |
| Mobile   | long click     | orientation switch | phone interrupt |
| Web      | mid click, right click | window size   | --- |

Widget actions are actions that are directly applied to specific widgets. We consider the widget image, the widget location, and the widget type presented in Section 3.1.1 as the target widget embedding feature. Whether a widget action is applicable on the current app page is determined by the widget type (inferred by the CNN model in Section 3.1.1). Different types of widget actions can be applied to one widget.

Page actions are actions applied directly to the app GUI page. The page actions mostly involve the page size or orientation changing. For example, the orientation switch action will change the app orientation, the window size action refers to the window size change. Besides, in order to simplify the action type, we fix some actions. For the swipe action, we only consider the up and down swipe directions. Therefore, we set a parameter of page actions as a number that identifies the size change ratio, or the orientations.

System actions refer to the actions that are applied to the operating system. Different from the widget and page actions, system actions are not directly related to the app GUI. No extra information is required to invoke the system actions.

---

Event” and “action” are equivalent in this paper.

**Table 3. Actions for PIRLTest**
Additionally, the actions are embedded to fit in the Q-network. Based on the analysis of the three categories of actions, action embedding is composed of three parts, the action itself, the target widget, and the parameter information.

For the action itself, we use a 17-dimension one-hot vector to represent the action type. For the target widget, we follow the processing in Section 3.1.1, and use an 8206-dimension vector to represent the target widget. For the window size parameter, we take the number parameter as a one-dimension vector to represent it. The three-part vectors are concatenated to obtain the action embedding vector. For some actions, the target widget and the parameter might not be applicable, and in such cases, we will pad the vector with "0"s in the corresponding places. The action embedding vector is in accordance with the RL framework of PIRLTest to make the action selection decisions.

For the action design, PIRLTest considers widget actions, pages actions, and system actions. These actions cover the vast majority of the actions that can be applied to different apps and corresponding GUI widgets. However, most existing learning-based automated testing methods only consider part of the actions, like the “click” action. Therefore, we provide a more complete action set to better explore more app states, improving the automated testing efficiency and effectiveness.

3.3 Reward Calculation

For each state transition \((s_t, a_t, s_{t+1})\), PIRLTest assigns a reward. A higher reward means a more significant transition, which may lead to an uncovered state, and then the RL framework learns the state and action values. A well-designed reward function can guide PIRLTest to explore more state space and avoid being stuck in a local optimum.

At the timestamp \(t\), the state transition is \((s_t, a_t, s_{t+1})\), and the transition times of \(s_t\) is \(N_t\), referring to the covered times of the state during the exploration; the executable action number in \(s_{t+1}\) is \(m_{t+1}\); and the executed action number is \(n_{t+1}\). We define the exploration rate of state \(s_{t+1}\) is \(e_{t+1} = \frac{n_{t+1}}{m_{t+1}}\), which means that how much percentage of the actions are executed. Based on the exploration rate, the reward is defined as follows:

\[
 r_t = \frac{1}{\sqrt{N_t}} \left( \frac{1}{\sqrt{N_t}} - \frac{1}{\sqrt{m_{t+1}}} \right)
\]  

(1)

Besides, the transition times value is added by 1 if the page similarity between the current page and one specific page in exploration memory is confirmed to be higher than a threshold. According to our introduction in Section 3.1, we calculate the page similarity with the widget extraction and layout characterization. With regard to the widget similarity, for all the widgets on page A \((p_A)\) and page B \((p_B)\), we obtain the one-to-one matched widget pairs if the widgets’ Interaction-over-Union (IoU) value \([18]\) is over the specific threshold (some widgets may not have matching widgets, and such widgets are neglected). For each matched widget pair, we calculate the distance with the embedded widget vector (Section 3.1.1) through the Euclidean distance. We define the average distance of all widget pairs as \(d(p_A, p_B)\), and the widget similarity is \(sim_w(p_A, p_B) = 1 - d(p_A, p_B)\), ranging in \([0, 1]\). The layout of a GUI page is represented by a string composed of widget nodes and curly braces (Fig. 5). We use tree edit distance \([53, 54]\) to calculate layout similarity between page A \(p_A\) and page B \(p_B\):

\[
 sim_l(p_A, p_B) = 1 - \frac{d(A, B)}{\max(n_A, n_B)}
\]  

(2)

where \(d(A, B)\) refers to the tree edit distance, and \(n_A, n_B\) refer to the node numbers of \(p_A\) and \(p_B\). Specifically, \(\max(n_A, n_B)\) actually indicates the theoretical maximum value of \(d(A, B)\). \(sim_l(p_A, p_B)\) ranges between \([0, 1]\). Page similarity is the weighted sum of widget and layout similarities. We initially set the 0.5-0.5 weight. We use
a threshold to determine whether two pages are the same. In this paper, we use 0.75 as default, following the common practice [73].

In this paper, we design a novel reward function in the RL framework, which comprehensively considers the exploration of both app GUI as a whole and the concrete GUI widgets. The design of the reward function corresponds to our app GUI state abstraction, including the GUI widget (microscopic perspective) and the GUI layout (macroscopic perspective). Compared with existing learning-based automated testing methods, which mostly consider the page similarity only, PIRLTest can more realistically reflect the app state situation, thus leading to the effective reward assignment and making the exploration effective and efficient.

3.4 Exploration Policy Execution

With the embedded current state and all the applicable actions on existing widgets of the app state, PIRLTest’s exploration policy calculates the values of such actions under such a state, and then executes the appropriate action, with the purpose of exploring as many un-explored states as possible. The exploration policy is a function that gives the execution probability distribution of all possible actions according to the current state. A better designed exploration policy can better guide the exploration of the RL framework on the apps under test.

For automated software testing, the content changes of every accessible app page or widget may cause the app state changes, so the state space to explore is huge. Moreover, the executable action set of different states is different, and the action space of the whole app to explore is huge. Therefore, traditional RL algorithms using the Q-Table will have difficulty processing such a situation. Considering the robust capability of deep learning models, we adopt a deep neural network in PIRLTest as the key strategy for exploring the app under test, which is supposed to effectively alleviate the explorative space explosion problem.

To determine a test event in the exploration, PIRLTest first merges the embedded state and each action to form the \( \langle \text{state}, \text{action} \rangle \) pair and then calculates the Q-value of the pair with the Q-network. We add different weights to the Q-values of various kinds of actions in order to reduce the probability of system actions and encourage more exploration within the app itself, and such a practice is widely adopted by literature [52, 76]. We set the weight of widget and page actions as 1, and the weight of system actions as 0.5. We set a lower weight for the system actions based on the common practices of existing approaches like Stoat [62] and Q-testing [52]. This practice can avoid executing system actions all the time but not exploring new GUI states. We try different weights from 0.1 to 0.9 at the interval of 0.1 on five apps as a pilot investigation, and we find that 0.5 is about to the optimal value. Therefore, we use the 0.5 as the weight of system actions when compared with the 1 for widget and page actions. With the weighed Q-values obtained, we use the SoftMax function to calculate the execution probability of each action and select which one to execute.

3.4.1 Q-Network

The Q-network is implemented with a fully connected neural network with four hidden layers, and each layer has 512 dimensions. The reason why using the fully connected neural network instead of a CNN or RNN model is that the fitting speed is much faster. During the exploration of the RL framework, the Q-network needs to be kept training with constantly refreshed training data, so the fitting speed is important to the efficiency. The fully connected neural network is capable of achieving a balance between high fitting speed and fitting accuracy. The input of the Q-network is the \( \langle \text{state}, \text{action} \rangle \) pair, which is the concatenated result of the embedded state and the embedded action. The output of the Q-network is a real number, indicating the Q-value of the given \( \langle \text{state}, \text{action} \rangle \) pair. Generally speaking, the Q-network takes a vector as the input. The vector is the concatenated state vector and action vector, then the vector goes through four hidden layers, where the parameters are determined by fitting the output Q-values and the expected Q-values of training data. The \( \langle \text{state}, \text{action} \rangle \) pair indicates applying an action on a specific app GUI state. The Q-value indicates the probability to transit to a new app GUI state of the applied action on the specific app GUI state. Then, the output layer is concatenated after the four hidden layers, which outputs a real number, indicating the Q-value of the
input \((\text{state, action})\) pair. The larger the Q-value is, the higher the probability of exploring new states will be. The loss function of the model is defined as the mean squared error of the output Q-value and the expected Q-value (calculated as Equation 3 in Section 3.4.2). When exploring the app under test, PIRLTest constantly re-trains the Q-network to make the fitting more accurate in evaluating the values of executing one specific action. The training process of the Q-network involves the state memory and is illustrated in Section 3.4.2. for PIRLTest, we train the Q-network to fit the Q-values of our newly designed app GUI states and a more complete action list in the RL framework. We do not use the Q-table because the states of the app require significantly increasing calculation overhead when the enumerated states increase to obtain the state similarity and exploration probability. Considering the features of GUI app state and state transition, especially the huge exploration space, we believe DQN (Q-network) is a suitable choice in the automated GUI testing scenario.

3.4.2 Exploration Memory. In order to better utilize the explored states to constantly adjust the Q-network, we design the exploration memory to record every state transition and the corresponding reward, which is the data source for the Q-network training. The rewards will have an effect on the exploration policy, and then guide the refreshing of the exploration policy. In the memory, the following data of timestamp \(t\) are recorded: the state transition \(\langle s_t, a_t, s_{t+1}\rangle\), the reward \(r_t\), and the action set \(A_{t+1}\) of state \(s_{t+1}\). \(s_t\), \(a_t\), and \(r_t\) represent the state embedding, action embedding, and the reward at the timestamp \(t\), respectively. \(A_{t+1}\) is the executable action set at the timestamp \(t + 1\). Such information can be used to calculate the \(Q_{target}^t\), which is the expected Q-value and can guide the fitting of the Q-network (Equation 3).

\[
Q_{target}^t = r_t + \gamma \max_{a \in A_{t+1}} Q (s_{t+1}, a)
\]  

Parameter \(\gamma\) is the decay rate, and a larger \(\gamma\) means that a reward of a more recent reward is more important. With the further exploration of PIRLTest, more and more data are recorded. Considering the efficiency problem, we cannot take all the data into the Q-network training. We have to select part of the data as the training data. In order to instruct the Q-network to tendentiously learn from recent state transitions, we use the most recent \(\frac{n}{2}\) data and randomly select another \(\frac{n}{2}\) data from the rest of data.

3.4.3 Randomization Strategy. PIRLTest can get the execution probability of all the actions under the current state with the values of the \((\text{state, action})\) pairs, and it is optimal to execute the action with the highest Q-value. However, if the optimal actions are always executed, the framework keeps exploring with existing knowledge, making it hard to explore unknown states. This is the so-called “exploration-exploitation dilemma” [57] in RL. For PIRLTest, we use the Boltzmann strategy [14] to alleviate the problem. The Boltzmann strategy assigns a probability to each \((\text{state, action})\) pair according to the Q-value. Then, the probabilities will directly affect which action is to be executed.

3.5 Implementation

In order to devote the proposed platform-independent reinforcement learning framework into practice, we design and implement the corresponding tool. In this section, we illustrate the concrete design and detailed algorithms and parameters. For the app GUI state abstraction, we use the VGG-16 model to extract the features. We follow the default training settings of hyper-parameters. The batch size is 256, the training epoch is 5, and the learning rate is 0.001. The training is conducted on a Ubuntu 20.04.2 machine with the GTX 1080Ti GPU. For the Q-network, we set the batch size as 64, the training epoch as 5, and the learning rate as 0.01, following the existing studies like Q-testing [52].

ACM Trans. Softw. Eng. Methodol.
4 EXPERIMENT

4.1 Experiment Setting

4.1.1 Research Question. We set three research questions to evaluate the PIRLTest, and the first two questions focus on the code coverage, bug detection capability, and corresponding cross analyses on mobile apps and web apps, respectively. The third question has an in-depth insight into the advantages of PIRLTest.

- RQ1 (Code Coverage): How effectively can PIRLTest cover app source code on different apps?
- RQ2 (Bug Detection): How effectively can PIRLTest detect real-world bugs on different apps?
- RQ3 (Advantage Analysis): How can PIRLTest outperform the baselines?

4.1.2 Experiment Subjects. For mobile apps, we collect real-world apps from existing studies, including [46, 52, 62]. Besides, we use widely-used open-source apps from a popular list [41]. For web apps, we use the apps in existing work like [76]. However, due to the maintenance problem of the apps, some apps cannot be compiled or instrumented, so we have to only use the usable ones. For the mobile apps, we first use a Samsung mobile device with Android 11, some apps are out-of-date, no longer maintained to be executed, and are not supported on this Android version, so we remove such apps. Then, due to the requirement to obtain code coverage and running logs, we use a widely used tool to intrude the apps, but some apps cannot be instrumented to obtain the code coverage, so we also remove such apps. Finally, we use 20 mobile apps in our evaluation. For the web apps, we consider the full source code, including front end and server end, to obtain the code coverage information, which is not used as the evaluation metric in existing studies, which only considers the HTML page coverage, so we choose the source-code-available apps in our evaluation. We respectively instrument the mobile apps with Jacoco\(^8\) and web apps with NCY\(^9\). More details of our experiment subject apps are available on our online package due to the page limit.

4.1.3 Baseline Approaches. In order to evaluate the capability of PIRLTest more comprehensively, we select 9 baselines for mobile app GUI testing and 2 baselines for web app GUI testing, including random-based ones, model-based ones, and learning-based ones. We use the Monkey [28] as a representative of random strategy. Monkey is originally used for mobile apps, so we implement the Monkey for the web platform. For the model-based strategy for mobile apps, we use four representative tools, Stoat [62], TimeMachine [23], ComboDroid [63], and Ape [31], as the baselines. These baselines are representative and state-of-the-art tools for mobile app exploration testing. For the learning-based strategy, we use four state-of-the-art approaches, the Humanoid [39], Q-testing [52], GPTDroid [42], and the WebExplor [76], which are cutting-edge technologies for automated mobile and web app testing. We believe such tools, which have proven their capability compared with other tools, are representative. With regard to the tool implementation, for Monkey on the mobile platform, the tool is directly usable provided by Google. For the Monkey on the web platform, we implement a tool imitating the strategy adopted by Google Monkey, guaranteeing that it can randomly generate test events for web app exploration. Some baselines, like the Stoat and Q-testing, are available because the authors open-source them, while some others are not available, like the GPTDroid, so we have to replicate them by ourselves, and we have tried our best to follow the paper descriptions to reproduce the tool. Experiments are repeated 5 times and we apply the Wilcoxon non-parametric statistical test with the consideration of the non-determinism of the algorithms. The average results are recorded when the p-value is less than 0.05.

4.1.4 Evaluation Metrics. Basically, we adopt the most widely accepted metric, coverage, as the main metric of this paper. Specifically, we utilize the coverage of two granularities, the line coverage and the branch coverage. We use the results of running 1.5 hours, 2 hours, 3 hours, and generate 500 test events, following the common

---

\(^8\)https://www.jacoco.org/jacoco/
\(^9\)https://istanbul.js.org/
practice like in [52, 62, 76]. One thing to notice is that if less than 500 events are generated within 1.5 hours, we will continue until 500 events are generated. During the comparison of 1.5-hour, 2-hour, and 3-hour results, we do not count how many events are generated. We evaluate the bug detection capability of PIRLTest on real-world apps by calculating how many bugs PIRLTest can detect. We identify crashes that are actually caused by the app under test by monitoring Logcat information, and we conduct the deduplication by analyzing the information regarding app activity, app package, and stack data. Further, we analyze the cross situation of code coverage and detected bugs, which means how many code lines/branches and detected bugs are covered by both tools or how many code lines/branches and detected bugs are additionally covered by one tool than that of another one. Specifically, cross bug detection is calculated according to the bug quality. The cross code coverage is denoted as $A \uparrow B$, which means the percentage of additional covered code of a tool to the all covered code of such a tool. $A \uparrow B$ is calculated as follows:

$$A \uparrow B = \frac{\text{coverage of tool A} - \text{coverage of tool B}}{\text{coverage of tool A}} \times 100\%$$ (4)

Table 4. Summary of Code Coverage & Bug Detection Capability Comparison: Average Code Coverage and Detected Bug Summation (M: Monkey, S: Stoat, T: TimeMachine, C: ComboDroid, H: Humanoid, A: APE, Q: Q-testing, G: GPTDroid, W: WebExplor, P: PIRLTest)

| Configuration          | Mobile App | Web App |
|------------------------|------------|---------|
|                        | M S T C H A Q G P | M W P |
| Line Coverage (500step) | 23.7%     | 23.7%   | 24.2% | 24.7% | 25.4% | 21.9% | 24.0% | 27.9% | 30.8% | 21.5% | 30.6% | 31.0% |
| Line Coverage (1.5h)   | 25.6%     | 25.5%   | 24.7% | 24.9% | 25.3% | 26.1% | 30.7% | 32.4% | 34.7% | 28.3% | 25.5% | 29.4% |
| Line Coverage (2h)     | 34.9%     | 35.4%   | 35.0% | 35.5% | 36.1% | 35.5% | 40.6% | 44.7% | 49.8% | 37.7% | 33.6% | 37.8% |
| Line Coverage (3h)     | 35.4%     | 35.9%   | 35.4% | 36.0% | 36.6% | 36.0% | 41.0% | 45.2% | 50.4% | 42.9% | 41.8% | 48.8% |
| Branch Coverage (500step) | 12.0% | 12.5%       | 13.1% | 13.5% | 14.7% | 14.7% | 12.4% | 16.2% | 16.9% | 12.7% | 18.7% | 19.2% |
| Branch Coverage (1.5h) | 14.0%     | 14.8%   | 15.5% | 15.7% | 15.8% | 16.5% | 18.6% | 18.9% | 19.7% | 12.6% | 10.8% | 13.2% |
| Branch Coverage (2h)   | 19.6%     | 20.2%   | 21.2% | 20.9% | 20.0% | 22.1% | 22.9% | 23.1% | 24.1% | 18.1% | 16.1% | 18.6% |
| Branch Coverage (3h)   | 22.9%     | 24.2%   | 24.4% | 23.8% | 24.3% | 24.6% | 26.8% | 25.7% | 28.9% | 19.4% | 17.0% | 19.7% |
| Detected Bug (500step) | 20        | 52      | 52    | 50    | 43    | 44    | 49    | 97    | 117   | 1     | 2     | 6     |
| Detected Bug (1.5h)    | 34        | 56      | 58    | 57    | 55    | 49    | 116   | 96    | 121   | 1     | 2     | 4     |
| Detected Bug (2h)      | 45        | 65      | 68    | 67    | 65    | 56    | 127   | 105   | 132   | 4     | 5     | 11    |
| Detected Bug (3h)      | 62        | 78      | 75    | 76    | 74    | 68    | 136   | 113   | 145   | 4     | 8     | 13    |

4.2 RQ1: Code Coverage

The results on code coverage are presented in Table 4, Fig. 6 and Fig. 7. According to the results, PIRLTest generally performs better than the baselines with regard to code coverage on both mobile apps and web apps. On mobile apps, PIRLTest achieves 7.2% - 40.7% more line coverage compared with different baselines after running 1.5 hours, 11.3% - 42.8% more line coverage after running 2 hours, and 11.4% - 42.2% more line coverage after running 3 hours. It achieves 4.4% - 41.4% more branch coverage compared with different baselines after running 1.5 hours, 4.4% - 22.5% more branch coverage after running 2 hours, and 7.9% - 26.3% more branch coverage after running 3 hours. On web apps, PIRLTest achieves 3.8% and 15.5% more line coverage compared with Monkey and WebExplor after running 1.5 hours, 0.4% and 12.5% more line coverage after running 2 hours, and 13.8% and 16.6% more line coverage after running 3 hours. It achieves 5.0% and 22.7% more branch coverage than the baselines after running 1.5 hours, 2.7% and 15.5% more branch coverage after running 2 hours, and 1.3% - 15.7% after running 3 hours.
The results indicate that PIRLTest can generate much fewer non-effective test events, while most test events generated by Monkey are non-effective, \textit{i.e.}, on blank areas. Learning-based approaches are more effective than random and model-based approaches due to the exploration strategy design. Besides, the baseline approach miss many valid GUI widgets that do not appear in the layout XML files, which is also how PIRLTest generates more effective test events based on the widget identification from the visual perspective. Regarding the test generation efficiency, learning-based approaches, including PIRLTest, have extra overhead in processing GUI pages and exploring with RL frameworks. This is also the reason why Monkey generates more test events, though most of which non-effective. Generally, learning-based approaches, including LLM-based ones, are better than non-learning-based approaches.

From the time cost aspect, we can find that the advantage of PIRLTest improves from 1.5-hour running to 3-hour running, we go through the test events and find that PIRLTest can generate longer test event sequences. For example, in the “OmniNotes” app, most functions are available only after logging in, while the entrance to login is not directly presented on the launching page, and a relatively long test event sequence is necessary to reach the login page, and then the exploration will continue after the login events. Given more time (1.5 hours to 2 hours, and to 3 hours), PIRLTest are more likely to generate longer test event sequences. This is also the reason that PIRLTest performs not so well as baselines on several apps, where the advantage of PIRLTest cannot be fully utilized.

Due to the test generation efficiency differences, The approaches are compared at the same test event quantity granularity. PIRLTest achieves 10.5% - 41.0% more line coverage compared with baselines on mobile apps. It achieves 4.1% - 41.0% more branch coverage than the baselines. On web apps, PIRLTest achieves 44.1% and 1.5%
Fig. 7. Code Coverage & Bug Detection Capability Comparison on Web Apps (M: Monkey, W: WebExplor, P: PIRLTest)

(a) Line Coverage (500step)  (b) Line Coverage (1.5h)  (c) Line Coverage (2h)  (d) Line Coverage (3h)
(e) Branch Coverage (500step)  (f) Branch Coverage (1.5h)  (g) Branch Coverage (2h)  (h) Branch Coverage (3h)
(i) Detected Bug (500step)   (j) Detected Bug (1.5h)   (k) Detected Bug (2h)   (l) Detected Bug (3h)

Fig. 8. Action Distribution in All Experiment Subjects

(a) Mobile App  (b) Web App
more line coverage compared with Monkey and WebExplor. It achieves 51.1% and 2.8% more branch coverage than the baselines. It is observed that the advantage of PIRLTest is more highlighted.

We investigate the actual generated test actions in our experiment, and the results are shown in Fig. 8. We can find that for both mobile and web apps, the “click” action takes over half of all generated actions, the reason is that most widgets are applicable for the “click” action. Except the “click” action, “input”, “return”, and “double click” are top three action types for both mobile and web apps. Other kinds of actions take a small percentage in all generated test events, which corresponds to the real app usage.

Table 5. Cross Coverage & Bug Detection Analysis (X refers to the baseline approaches, M: Monkey, S: Stoat, T: TimeMachine, C: ComboDroid, H: Humanoid, A: APE, Q: Q-testing, G: GPTDroid, W: WebExplor, P: PIRLTest.

| Configuration          | Mobile App | Web App |
|------------------------|------------|---------|
|                        | M S T C H A Q G | M W |
| Line Coverage (500step) | P ↑ X 19.6% 27.6% 12.8% 22.1% 19.5% 12.8% 20.1% 19.3% | 4.3% 12.0% |
|                        | X ↑ P 5.3% 9.8% 9.4% 8.5% 7.8% 10.7% 7.3% 7.1% | 2.7% 1.8% |
| Line Coverage (1.5h)   | P ↑ X 24.2% 24.6% 20.0% 23.4% 28.6% 28.3% 13.7% 13.6% | 16.0% 36.7% |
|                        | X ↑ P 5.5% 6.6% 8.5% 10.0% 8.1% 11.2% 6.7% 6.2% | 20.0% 0.7% |
| Line Coverage (2h)     | P ↑ X 20.0% 30.2% 15.5% 17.7% 15.4% 8.8% 22.8% 15.8% | 12.7% 16.6% |
|                        | X ↑ P 2.6% 9.3% 5.6% 9.2% 6.3% 14.5% 11.6% 11.2% | 5.8% 3.4% |
| Line Coverage (3h)     | P ↑ X 27.7% 29.6% 16.3% 25.4% 25.4% 31.5% 15.3% 12.9% | 16.2% 39.2% |
|                        | X ↑ P 2.8% 8.9% 4.9% 8.3% 6.1% 11.6% 3.0% 3.0% | 6.8% 4.7% |
| Branch Coverage (500step) | P ↑ X 27.1% 53.7% 51.5% 52.4% 54.3% 56.8% 35.4% 34.2% | 22.3% 24.2% |
|                        | X ↑ P 9.6% 35.7% 14.8% 21.1% 14.5% 12.7% 21.5% 19.8% | 21.3% 7.3% |
| Branch Coverage (1.5h) | P ↑ X 43.1% 21.8% 33.5% 32.5% 39.0% 37.6% 33.7% 29.2% | 18.8% 49.6% |
|                        | X ↑ P 12.6% 21.9% 11.9% 16.1% 12.1% 11.0% 14.5% 12.9% | 26.4% 14.9% |
| Branch Coverage (2h)   | P ↑ X 22.3% 52.1% 54.5% 55.1% 55.6% 57.8% 57.3% 55.9% | 26.7% 22.3% |
|                        | X ↑ P 7.7% 8.8% 12.7% 15.7% 13.3% 14.5% 29.2% 27.3% | 19.3% 6.7% |
| Branch Coverage (3h)   | P ↑ X 43.1% 23.2% 25.6% 27.2% 31.7% 35.4% 34.2% 25.2% | 22.7% 48.9% |
|                        | X ↑ P 15.7% 21.3% 14.8% 21.1% 14.5% 12.7% 21.5% 19.8% | 21.3% 7.3% |

Detected Bug (500step) | P ↑ X 102 102 100 96 94 89 97 96 | 5 4 |
|                        | X ↑ P 15 15 17 21 23 28 20 21 | 1 2 |
|                        | 5 37 35 29 20 16 29 76 | 0 0 |

Detected Bug (1.5h) | P ↑ X 97 106 103 100 93 92 55 90 | 3 2 |
|                        | X ↑ P 24 15 18 21 28 29 66 31 | 1 2 |
|                        | 10 41 40 36 27 20 50 65 | 0 0 |

Detected Bug (2h) | P ↑ X 109 110 101 108 109 101 87 71 | 8 7 |
|                        | X ↑ P 23 22 31 24 23 31 45 61 | 3 4 |
|                        | 22 43 37 43 42 25 82 44 | 1 1 |

Detected Bug (3h) | P ↑ X 114 122 107 110 116 114 80 96 | 9 7 |
|                        | X ↑ P 31 23 38 35 29 31 65 49 | 4 6 |
|                        | 31 55 37 41 45 37 71 64 | 0 2 |

To further illustrate the advantage of PIRLTest, we conduct the cross analysis on code coverage. The results are shown in Table 5.

For the 500-step results on mobile apps, the baselines can only cover 5.3% - 10.7% more code lines and 9.6% - 35.7% more code branches than PIRLTest. However, PIRLTest can cover 12.8% - 27.6% more code lines and 27.1% - 56.8% more code branches than baselines in contrast. For the 1.5-hour results on mobile apps, the baselines can
only cover 5.5% - 11.2% more code lines and 11.0% - 21.9% more code branches than PIRLTest. However, PIRLTest can cover 13.6% - 28.6% more code lines and 21.8% - 43.1% more code branches than baselines in contrast. For the 500-step results on web apps, the baselines can only cover 1.8% - 2.7% more code lines and 7.3% - 21.3% more code branches than PIRLTest. However, PIRLTest can cover 4.3% - 12.0% more code lines and 22.3% - 24.2% more code branches that are not covered by the baselines. For the 1.5-hour results on web apps, the baselines can only cover 0.7% - 20.0% more code lines and 14.9% - 26.4% more code branches than PIRLTest. However, PIRLTest can cover 22.3% - 24.2% more code lines and 18.8% - 49.6% more code branches that are not covered by the baselines. The data is similar for the 2-hour and 3-hour results, and PIRLTest can actually cover much different code that cannot be covered by the baselines. Although the advantage of PIRLTest in absolute coverage information might not be quite outstanding, the results on cross-coverage analysis show that PIRLTest can cover much code that cannot be covered by the state-of-the-art baseline approaches, which are obvious on both mobile apps and web apps. The results elaborate on the excellent performance of PIRLTest in covering more app source code.

4.3 RQ2: Bug Detection

In addition to code coverage, bug detection capability is more direct evidence to show the effectiveness of PIRLTest. The results in Table 4, Fig. 6 and Fig. 7 indicate that PIRLTest can detect more bugs compared with the baseline approaches.

For the bug detection capability, PIRLTest detects 5 - 87, 5 - 87, and 9 - 83 more bugs than the baselines within 1.5 hours, 2 hours, and 3 hours, respectively, on mobile apps. Besides, PIRLTest detect 20 - 97 more bugs than the baselines within 500 test events on mobile apps. PIRLTest detects 2 - 3, 6 - 7, and 5 - 9 more bugs than the baselines within 1.5 hours, 2 hours, and 3 hours, respectively, on web apps. PIRLTest detects 4 - 5 more bugs within 500 test events on web apps. We review the detected bugs and confirm that all the bugs are crash bugs. We have reported the bugs detected by PIRLTest to the developers, and 21 bugs are confirmed or fixed.

The cross analysis is conducted on the detected bugs (Table 5). For mobile apps, PIRLTest detects 55 - 106, 71 - 110, and 80 - 122 bugs that the baselines cannot detect within 1.5 hours running, 2 hours running, and 3 hours running, respectively. PIRLTest detects 89 - 102 bugs over baselines within 500 test events. For web apps, PIRLTest detects 2 - 3, 3 - 4, and 7 - 9 bugs that the baselines cannot detect within 1.5 hours running, 2 hours running, and 3 hours running, respectively. PIRLTest detects 4 - 5 bugs over baselines within 500 test events. The bugs uniquely detected by PIRLTest are much more than the bugs uniquely detected by baselines. The results indicate that PIRLTest can detect more bugs that cannot be detected by the baseline approaches. The results come from the design of GUI state abstraction, which help better characterize the GUI and corresponding widgets to encourage effective exploration. Another noteworthy observation is that PIRLTest can detect more bugs with higher efficiency (within 500 test events), especially compared with Q-testing on mobile apps.

Generally speaking, the correlation relationship between the code coverage and bug detection capability is positive based on the experiment results of different time costs (1.5 hours, 2 hours, and 3 hours), which means that the more code one approach can cover, the more likely it can detect more bugs. However, the correlation relationship is also affects by other factors. For example, the distribution of bugs within the whole app is unknown, and it is hard to assign specific targets to automated GUI testing approaches to detect bugs. Second, different inputs can lead to different. Due to the weak correlation relationship between the code coverage and bug detection capability, we present the two metrics to better illustrate the performance of PIRLTest.

The data show that PIRLTest has an excellent performance in detecting bugs on both mobile and web platforms. Specifically, the cross analysis on detected bugs proves that PIRLTest is capable of detecting bugs that are hard to be detected by the baseline approaches. We have an in-depth analysis in the following section.

ACM Trans. Softw. Eng. Methodol.
4.4 RQ3: Advantage Analysis
We have an in-depth investigation of the results of PIRLTest over the baselines, and we list three of the significant reasons that help improve the effectiveness of PIRLTest.

**Widget Granularity Exploration.** The widget-based RL exploration is one of the most significant novel contributions of this paper. Instead of retrieving the action targets from the layout XML files like some existing approaches, PIRLTest directly starts from the GUI page and obtains more effective widgets to alleviate the problems presented in Section 2.3. This is one of the significant reasons for the success of PIRLTest. However, the widget accuracy [18] is one of the obstacles that hinder PIRLTest from achieving a better performance.

**Q-network with State and Action Embedding.** After getting the effective embedding to the states and the actions, the Q-value of one specific state-action pair will be refreshed, so the GUI pages with similar appearances that carry similar functions will be recognized to avoid repeatedly covering the same code snippets. The design of the embedding and the Q-network gives PIRLTest the capability of processing the huge exploration state and action space. However, there exist a few situations where the assumption that similar GUI layouts are similar pages may not work, so the Q-network may under-estimate the values of such pages.

**Long Semantic Action Sequence.** This point is the most significant one in our opinion, which is that PIRLTest shows the potential of generating a long action sequence within the context. During the generation of long semantic action sequences, it may be interrupted if some irrelevant actions are generated. Given that the "future" reward is considered in the value estimation of current actions, a longer action sequence should be preferred by the RL exploration mechanism, and such a long action sequence tends to have context semantics from the perspective of human testers.

In this paper, we implement the PIRLTest and conduct an empirical evaluation on Android and web platforms. However, we hold that PIRLTest is a platform-independent approach, and will be effective in all GUI-based platforms, including iOS, harmonyOS, desktop operating systems, or even some embedded systems. The only accordance of PIRLTest is the analysis of the widget extraction and the layout characterization. As long as the platform has a GUI-based user interface, PIRLTest will be capable of exploring the state and action space. We believe PIRLTest has strong generalizability on different platforms and behaves at zero cost when adapting different platforms.

Another point is about efficiency. PIRLTest seems to have relatively poor efficiency compared with random-based approaches, i.e., Monkey. However, this is the common problem of model-based approaches (based on dynamic analysis) or learning-based approaches. As far as we know, none of the existing work has a comparison of the action generation efficiency. We hold that in the future, PIRLTest will be a complement to the random-based approaches, and will cover more code that cannot be covered by the existing approaches.

4.5 Threats to Validity
The main threat may be the **app selection.** In order to reduce this threat, we use the app under experiment from the existing research, including [46, 52, 62, 76]. We use widely-used open-source apps from a popular list [41] because some apps from the above papers are not accessible due to maintenance problems. Besides, the used apps cover many different categories, which expand the scenarios of GUI layout styles, ensuring the generalizability of PIRLTest.

Another threat may be the **parameter or hyperparameter settings,** like the thresholds in the widget recognition algorithm, or the deep learning model training. In order to alleviate the bias, we set the parameters according to domain knowledge or following the common practice from existing work. The training data are fully isolated from the experiment subjects we use in the evaluation. For some critical parameters, we conduct pilot studies to find suitable settings before the evaluation. For the baselines, we use the default settings or the open-source package to avoid possible biases.
PIRLTest contains several deep learning models, and the performance of the proposed deep learning networks may be a potential threat. In order to mitigate this threat, we try our best to collect more data to train the neural networks. We use networks with simple structures to avoid performance problems that may affect the effectiveness of PIRLTest.

4.6 Discussion
Compared with the baselines, PIRLTest has achieved promising results. However, there are some limitations for the proposed approach, which can also be solved in future work. First, due to the special design of app GUI state abstraction, PIRLTest can only process apps with table-like or list-like layout. Although such layout takes a vast majority percentage of all kinds of apps, there are few categories of apps PIRLTest cannot process, like the video games. However, with the development of vision-based GUI processing technologies, we believe the widget extraction and layout characterization of app GUI can be advanced. Second, although PIRLTest outperforms the baseline approaches, including the LLM-based tool, we believe it can be more effective to combine both, taking advantages of the app understanding capability of LLMs and exploration reward mechanism of RL frameworks. Third, although PIRLTest shows preliminary capabilities of generating long semantic action sequences, we believe the potentials of combining app business logic is not fully explored. Integrating the domain knowledge of human testers may further advance the capability of RL frameworks.

5 RELATED WORK
5.1 Automated Software Exploration
The most basic strategy of automated software testing is the random-based strategy. Among them, Monkey is the most widely used tool [28], and performs quite well on specific benchmark apps [19]. Monkey is capable of generating a large number of test events with high efficiency. However, the shortcomings are obvious. Due to the lack of effective guidance, Monkey generates a large percentage of non-effective or redundant test events, posing a threat to the testing effectiveness. Such a problem is alleviated by restricting the GUI states [22], but the exploration process still lacks guidance.

Approaches adopting model-based strategy [3, 70] build specific models for the testing exploration with static or dynamic (or combination) app analysis [48]. AndroidRipper [2] proposed by Amalfitano et al. adopts a user-interface driven ripper to explore the app GUI to exercise the app in a structured way. Baek et al. [7] present the GUICC, which provides the selection of multiple abstraction levels for GUI model generation with a set of multi-level GUI comparison criteria. Aimdroid [30] introduced by Gu et al. is a tool that aims to manage the exploration of activities and to minimize unnecessary transitions between them through an activity-insulated multi-level strategy during the testing. Biagiola et al. propose SubWeb [10], which takes advantage of the navigation structure implicitly specified by developers for web testing. DIG [11] pre-selects the most promising candidate test cases based on their diversity from previous tests. GoalExplorer [37] firstly statically models the app GUI and transitions, and then guides the dynamic exploration of the app to the particular target of interest. Sapienz [46], proposed by Mao et al., is a typical tool that uses multi-objective search-based testing to automatically explore and optimize test sequences. Stout [62] is a representative and representative model-based tool, which uses a stochastic Finite State Machine model to describe the behavior of AUT. Due to the limitation of the modeling algorithms, the model-based approach can neither precisely nor completely characterize the apps under test.

Benefiting from the development of deep learning and machine learning models, learning-based automated testing technologies also emerge. White et al. [67] propose a machine learning-based technique to improve GUI testing by automatically identifying GUI widgets in screenshots. Li et al. [39] propose Humanoid, an automated black-box Android app testing tool based on deep learning models. Reinforcement learning is one group of suitable algorithms for app exploration. Koroglu et al. [36] propose a fully automated black-box testing methodology,
which explores GUI actions using Q-Learning. Zhao et al. [75] propose DinoDroid, an approach based on deep Q-networks to automate testing of Android apps. Lv et al. [44] introduce an automated model-based GUI testing technique to accelerate the testing process and experiments on two popular industrial apps show its effectiveness. Wuji [77] is a typical approach that utilizes RL algorithm to explore video game apps. It balances winning the game and exploring the space of the game to uncover more bugs. Q-testing is an effective tool for Android testing [52] using the RL algorithm, it trains a neural network to divide different states at the granularity of functional scenarios. WebExplor [76] is an advanced tool for web app testing with the RL algorithm. It adopts a curiosity-driven model to generate high-quality test events containing temporal logical relations. Romdhana et al. [58] compare different RL algorithms in mobile app testing.

Among the aforementioned automated testing approaches, much progress is made to improve the automated software testing effectiveness and efficiency. However, as claimed in Section 2.3, current approaches are still faced with critical problems, which hinder the approaches from showing better performance. Besides, as far as we know, none of the current approaches conduct platform-independent app testing, which builds a high bar for app developers when faced with the sharply growing number of running platforms.

5.2 App GUI Understanding

Due to the fragmentation problem of apps [66], more and more work starts to apply GUI understanding technologies to assist the software engineering tasks [73]. One group of researchers uses GUI understanding technologies to conduct reverse engineering and generate code snippets from GUI images [15, 50, 51, 74]. Such tools adopt traditional computer vision technologies or deep learning/machine learning models to analyze the GUI images.

Another important direction of using GUI understanding in software testing is test report analysis and optimization. Yu et al. [71] use image understanding technologies to generate reports for app screenshots indicating bugs. Yu et al. [72] prioritize crowdsourced test reports with deep image and text semantic understanding. Besides, representative work like [25, 26, 64] all use GUI image understanding in crowdsourced test report optimization. Studies [9, 24] have been proposed to help reproduce the video-based test reports by analyzing the GUI information in the video test reports. Tango [21] is capable of detecting duplication in video-based test reports. Also, GUI understanding is widely used in record and replay test scripts. Behrang et al. [8] introduce AppTestMigrator to migrate test cases between apps using the similarity among GUI widgets. Qian et al. [55] present a tool that uses CV algorithms to recognize GUI widgets and then controls robots to complete the automated testing tasks. Yu et al. [73] realize the cross-platform record and replay of test scripts with the image and layout characterization. One significant basis for applying GUI understanding technologies in software testing is widget identification. Chen et al. discuss the widget recognition algorithms in [18]. Xiao et al. propose IconIntent [69] to understand the intents of GUI widgets to identify sensitivity. Liu et al. propose a tool to help detect UI display issues with deep GUI understanding [43]. Chen et al. [17] propose a deep learning model to help predict the labels of specific GUI widgets. Chen et al. [16] introduce a GUI design search engine to assist GUI designers.

The above studies inspire us to start automated testing from the GUI perspective with computer vision technologies. Moreover, considering the nature that some special widgets, like embedded Canvas elements or other customized elements, or the embedded HTML pages, cannot be retrieved from GUI layout files with traditional widget identification approaches, the visual analysis on app GUI improves the effectiveness of automated software testing.

6 CONCLUSION

To tackle the challenges of automated explorative testing, this paper proposes an effective, platform-independent GUI testing framework with image embedding and reinforcement learning. PIRLTest integrates a novel algorithm to embed the app GUI for the state comparison in the RL algorithm and uses a deep neural network to construct
the Q-network for the state-action value determination. PIRLTest is the first approach that bridges the gap of app exploration on different platforms. The experiment results show that PIRLTest can cover 6.3–41.4% more code on mobile apps and 1.5–51.1% more code on web apps, with much of the code not covered by the baselines, and it can detect 128 unique bugs on mobile and web apps, including 100 ones that are not detected by the baselines. PIRLTest is the first approach to independently test apps of different platforms with excellent effectiveness.

ACKNOWLEDGMENTS

The authors would like to thank the editors and anonymous reviewers for their time and comments. This work is supported partially by the National Natural Science Foundation of China (62141215, 61932012, 62372228), and the Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

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. 2–8.

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

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfario 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] Appium. 2022. http://appium.io/.

[5] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability 24, 3 (2014), 219–250.

[6] Snigdha Athaiya and Raghavan Komandoor. 2017. Testing and analysis of web applications using page models. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 181–191.

[7] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 238–249.

[8] Farnaz Behrang and Alessandro Orso. 2018. Test migration for efficient large-scale assessment of mobile app coding assignments. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. 164–175.

[9] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 309–321.

[10] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. 2017. Search based path and input data generation for web application testing. In Proceedings of the 2017 International Symposium on Search Based Software Engineering. Springer, 18–32.

[11] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based web test generation. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 142–153.

[12] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A Efros. 2019. Large-Scale Study of Curiosity-Driven Learning. Proceedings of the 7th International Conference on Learning Representations (2019).

[13] John Canny. 1986. A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence 6 (1986), 679–698.

[14] Nicolò Cesa-Bianchi, Claudio Gentile, Gábor Lugosi, and Gergely Neu. 2017. Boltzmann exploration done right. Advances in neural information processing systems 30 (2017).

[15] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI design image to GUI skeleton: a neural machine translator to bootstrap mobile GUI implementation. In Proceedings of the 40th International Conference on Software Engineering. 665–676.

[16] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI design search through image autoencoder. ACM Transactions on Software Engineering and Methodology 29, 3 (2020), 1–31.

[17] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering. IEEE, 322–334.

[18] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: old fashioned or deep learning or a combination?. In Proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 1202–1214.

ACM Trans. Softw. Eng. Methodol.
Effective, Platform-Independent GUI Testing via Image Embedding and Reinforcement Learning • 25

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

[20] Eliane Collins, Arilo Neto, Auri Vincenzi, and José Maldonado. 2021. Deep reinforcement learning based Android application GUI testing. In Proceedings of the XXXV Brazilian Symposium on Software Engineering. IEEE, 186–194.

[21] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshlyvanyk. 2021. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering. IEEE, 957–969.

[22] Faraz Yazdani Banafshe Daragh and Sam Malek. 2021. Deep GUI: Black-box GUI Input Generation with Deep Learning. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 905–916.

[23] Zhen Dong, Marcel Bohme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 481–492.

[24] Sidong Feng and Chunyang Chen. 2022. GIFdroid: Automated Replay of Visual Bug Reports for Android Apps. In Proceedings of the 2022 IEEE/ACM 43rd International Conference on Software Engineering. IEEE.

[25] Yang Feng, Zhenyu Chen, James A Jones, Chunrong Fang, and Baowen Xu. 2015. Test report prioritization to assist crowdsourced testing. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 225–236.

[26] Yang Feng, James A Jones, Zhenyu Chen, and Chunrong Fang. 2016. Multi-objective test report prioritization using image understanding. In Proceedings of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering. IEEE, 202–213.

[27] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In Proceedings of the 2013 35th International Conference on Software Engineering. IEEE, 72–81.

[28] Google. 2022. https://developer.android.com/studio/test/other-testing-tools/monkey.

[29] Alex Graves. 2012. Long short-term memory. In Supervised sequence labelling with recurrent neural networks (2012), 37–45.

[30] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaozing Ma, and Jian Lu. 2017. Aimdroid: Activity-insulated multi-level automated testing for android applications. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution. IEEE, 103–114.

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

[32] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. Sara: self-replay augmented record and replay for android in industrial cases. In Proceedings of the 28th acm sigsoft international symposium on software testing and analysis. 90–100.

[33] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 215–224.

[34] Seunghui Jang, Ki Yong Lee, and Yanggon Kim. 2020. An Approach to Improving the Effectiveness of Data Augmentation for Deep Neural Networks. In 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 1290–1295.

[35] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. In Journal of artificial intelligence research 4 (1996), 237–285.

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

[37] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for android applications. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 115–127.

[38] Cong Li, Yanyan Jiang, and Chang Xu. 2022. Cross-device record and replay for Android apps. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 395–407.

[39] 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.

[40] Zhen Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lu. 2019. Boosting operational dnn testing efficiency through conditioning. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 499–509.

[41] Pei Liu, Li Li, Yanjie Zhao, Xiaoyu Sun, and John Grundy. 2020. Androzooopen: Collecting large-scale open source android apps for the research community. In Proceedings of the 37th International Conference on Mining Software Repositories. 548–552.

[42] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhao Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing. arXiv preprint arXiv:2305.09434 (2023).

[43] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 398–409.
[44] Zhewei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In 37th IEEE/ACM International Conference on Automated Software Engineering, 1–5.

[45] Aravind Machiry, Rohan Tahliali, 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, 224–234.

[46] 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, 94–105.

[47] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. ACM Transactions on the Web 6, 1 (2012), 1–30.

[48] Ali Mesbah, Arie Van Deursen, and Danny Roest. 2011. Invariant-based automatic testing of modern web applications. IEEE Transactions on Software Engineering 38, 1 (2011), 35–53.

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

[50] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. IEEE Transactions on Software Engineering 46, 2 (2018), 196–221.

[51] Tuan Anh Nguyen and Christoph Csalnner. 2015. Reverse engineering mobile application user interfaces with renau (i). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 248–259.

[52] 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, 153–164.

[53] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient computation of the tree edit distance. ACM Transactions on Database Systems (TODS) 40, 1 (2015), 1–40.

[54] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. Information Systems 56 (2016), 157–173.

[55] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. Roscript: a visual script driven truly non-intrusive robotic testing system for touch screen applications. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 297–308.

[56] Eric Steven Raymond. 2003. Applying the Rule of Least Surprise. The Art of Unix Programming (2003), 525.

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

[58] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2021. Deep Reinforcement Learning for Black-Box Testing of Android Apps. ACM Transactions on Software Engineering and Methodology (nov 2021). https://doi.org/10.1145/3502868 Just Accepted.

[59] Peter Seebach. 2001. The cranky user: The principle of least astonishment. IBM DeveloperWorks (2001).

[60] Selenium. 2022. https://www.selenium.dev/.

[61] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In Proceedings of the 3rd International Conference on Learning Representations, 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.

[62] 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, 245–256.

[63] Jieue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 469–480.

[64] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don’t lie: Duplicate crowdtesting reports detection for Android Enhanced by Reinforcement Learning. In Proceedings of the 2019 31st IEEE/ACM International Conference on Automated Software Engineering, 226–237.

[65] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In Proceedings of the 24th ACM SIGSOFT International Symposium on Software Testing and Analysis, 307–317.

[66] Chelsea C White III and Douglas J White. 1989. Markov decision processes. European Journal of Operational Research 39, 1 (1989), 1–16.

[67] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. Iconintent: automatic identification of sensitive ui interface state changes. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 1277–1279.

[68] Shengcheng Yu, Chunrong Fang, Zhenfei Cao, Xu Wang, Tongyu Li, and Zhenyu Chen. 2021. Prioritize crowdsourced test reports via deep screenshot understanding. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering, IEEE, 946–956.
[73] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and image recognition driving cross-platform automated mobile testing. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering. IEEE, 1561–1571.

[74] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. GIUGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering. IEEE, 748–760.

[75] Yu Zhao, Brent Harrison, and Tingting Yu. 2022. DinoDroid: Testing Android Apps Using Deep Q-Networks. arXiv preprint arXiv:2210.06307 (2022).

[76] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic web testing using curiosity-driven reinforcement learning. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering. IEEE, 423–435.

[77] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 772–784.