A Unified Framework for Mini-game Testing: Experience on WeChat

Chaozheng Wang
The Chinese University of Hong Kong
Hong Kong, China
adf111178@gmail.com

Haochuan Lu
Tencent Inc.
Shenzhen, China
hudsonhclu@tencent.com

Cuiyun Gao*
The Chinese University of Hong Kong
Hong Kong, China
cuiyungao@outlook.com

Zongjie Li
Hong Kong University of Science and Technology
Hong Kong
zligo@cse.ust.hk

Ting Xiong
Tencent Inc.
Shenzhen, China
candyxiong@tencent.com

Yuetang Deng
Tencent Inc.
Shenzhen, China
yuetangdeng@tencent.com

ABSTRACT

Mobile games play an increasingly important role in our daily life. The quality of mobile games can substantially affect the user experience and game revenue. Different from traditional mobile games, the mini-games provided by our partner, *Tencent*, are embedded in the mobile app *WeChat*, so users do not need to install specific game apps and can directly play the games in the app. Due to the convenient installation, *WeChat* has attracted large numbers of developers to design and publish on the mini-game platform in the app. Until now, the platform has more than one hundred thousand published mini-games. Manually testing all the mini-games requires enormous effort and is impractical. There exist automated game testing methods; however, they are difficult to be applied for testing mini-games for the following reasons: 1) Effective game testing heavily relies on prior knowledge about game operations and extraction of GUI widget trees. However, this knowledge is specific and not always applicable when testing a large number of mini-games with complex game engines (e.g., Unity). 2) The highly diverse GUI widget design of mini-games deviates significantly from that of mobile apps. Such issue prevents the existing image-based GUI widget detection techniques from effectively detecting widgets in mini-games.

To address the aforementioned issues, we propose a unified framework for black-box mini-game testing named *iExplorerGame*. iExplorerGame involves a mixed GUI widget detection approach incorporating both deep learning-based object detection and edge aggregation-based segmentation for detecting GUI widgets in mini-games. A category-aware testing strategy is then proposed for testing mini-games, with different categories of widgets (e.g., sliding and clicking widgets) considered. iExplorerGame has been deployed in WeChat for more than six months. In the past 30 days, iExplorerGame has tested large-scale mini-games (i.e., 76,000) and successfully found 22,144 real bugs.

CCS CONCEPTS

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

KEYWORDS

game testing, GUI widget detection

ACM Reference Format:
Chaozheng Wang, Haochuan Lu, Cuiyun Gao, Zongjie Li, Ting Xiong, and Yuetang Deng. 2023. A Unified Framework for Mini-game Testing: Experience on WeChat. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3613868

1 INTRODUCTION

Mobile games have become increasingly popular among smartphone users in the past few decades. According to the work [1], mobile game spending has achieved 110 billion dollars across app stores in 2022. The huge profits have spawned not only a growing number of games but also multiple mini-game platforms which provide numerous mini-games. Different from traditional mobile game apps, mini-games are embedded in mobile apps, so that users can directly play the games without downloading additional game apps. This "plug-and-play" feature has attracted numerous players, as well as many developers and development platforms. One of the most popular mini-game platforms worldwide is released by our partner, *Tencent*, and contained within the app *WeChat*. Until now, WeChat mini-game platform has been successful in attracting hundreds of thousands of developers and currently hosts over 100,000 mini-games. This has resulted in a user base of over one billion for mini-games and 400 million monthly active users.

In the WeChat mini-game platform, developers build mini-games based on the mini-game framework [5] provided by WeChat or game engines (e.g., LayaAir [2] and Unity [4]) supported by the WeChat mini-game platform. To provide high-quality mini-game

*Corresponding author. The author is also affiliated with Peng Cheng Laboratory and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3613868

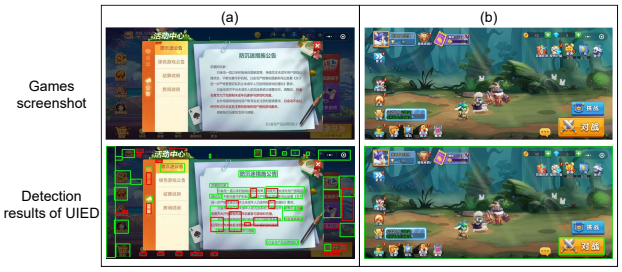

Figure 1: Examples of GUI widget detection results of UIED [63] for mini-games. Red and green boxes denote text and on-text widgets detection results.

service to users, it is necessary for WeChat to perform platform testing and auditing. Although manually testing the mini-games can accurately pinpoint bugs, it is inefficient and impractical due to the large number of mini-games. There exist approaches in automating mobile app testing [42, 52] or specific game testing [70]. However, they can hardly be applied to mini-game testing due to the following issues.

(1) **Lack of prior knowledge of mini-games**. Existing research on game testing [70] requires testing games in white box manners, e.g., they rely on involved operations and GUI widget trees of the games. However, the prior knowledge of the 120 thousand third-party mini-games published in WeChat is inaccessible so it is impossible to design specific testing strategies for each game. Besides, due to the complex game engines such as Unity [4], the GUI widget trees from mini-game pages are hard to be automatically extracted. Without the GUI widget information (e.g., widget location and type), the existing game testing techniques are not able to conduct game operations.

(2) **Diverse GUI design in mini-games.** Image-based testing can address the limitation of lacking prior knowledge for mini-games, which first detects GUI widgets on the game screenshots and then performs widget-based testing. There exists research on GUI widgets for Android and Web apps [21, 63]. However, these techniques are not applicable in mini-game testing because the GUI design of mini-games is much more complex and diverse, which differs greatly from that of Android and Web apps. As Figure 1 shows, the popular app GUI widget detection technique UIED [63] can neither recognize the buttons on the game screenshots (e.g., UIED fails to detect the majority of buttons except the yellow one in Figure 1 (b)) nor produce precise prediction (e.g., a huge number of invalid detected widgets on the background as shown in Figure 1 (a)). Without accurately detecting the GUI widgets, we can only cover few mini-game pages, leading to poor testing performance.

In this paper, we propose an image-based mini-game testing approach *iExplorerGame* to address the aforementioned issues. First, we propose a mixed GUI widget detection approach involving both deep learning-based object detection (DLOD) and edge aggregation-based segmentation (EAS) for detecting GUI widgets in mini-games. Particularly, to utilize deep learning models for GUI widget detection, we manually label a GUI widget detection dataset for mini-games, which contains 6,024 game screenshots and covers 343 mini-games. To facilitate mini-game testing, we classify the widgets into five categories including *click*, *target*, *slide*, *popup*, and *checkbox*. Combining DLOD and EAS, our iExplorerGame can automatically detect and recognize the GUI widgets in the mini-game screenshots without having to access GUI widget trees. Second, we design a category-aware testing strategy for large-scale mini-game testing based on GUI widget detection results. We assign a feature vector for each detected widget and compute priority. To evaluate the testing effectiveness of iExplorerGame for large-scale mini-games in WeChat, we deploy the iExplorerGame into WeChat automated game testing platform. Experiments demonstrate that our iExplorerGame significantly outperforms previous methods in detecting GUI widgets for mini-games (e.g., achieving 0.7 MAP@95 in detecting clickable widgets). Moreover, Extensive evaluation on 319 mini-games shows that iExplorerGame can explore 44 pages on average within ten minutes. In the past 30 days, iExplorerGame has successfully found 22,144 mini-game bugs including 17,677 crash and 4,467 JavaScript error bugs in 76,000 tested mini-games. The found bugs have been confirmed and feedback to developers.

In summary, this paper makes the following contributions:

- We build an image-based approach iExplorerGame for black-box mini-game testing. To the best of our knowledge, we are the first to explore large-scale black box mini-game testing.
- We conduct extensive experiments showing that our mixed GUI widget detection can significantly outperform previous work in detecting GUI widgets for mini-games.
- We have deployed iExplorerGame to test the thousands of games published on our WeChat mini-game platform.

Paper structure. The remainder of the paper is organized as follows. Section 2 describes our proposed iExplorerGame. Section 3 and 4 introduce our experiment setup and results. Section 5 discusses the threat to validity and Section 6 introduces related work.

2 IEXPLORERGAME

In this section, we will introduce the details of our proposed iExplorerGame. The overview of iExplorerGame is shown in Figure 2. iExplorerGame consists of two parts including *a mixed GUI detection approach* and *category-aware testing strategy*. The iExplorerGame takes a screenshot of the testing mini-game as input, and detects potential GUI widgets via deep learning-based object detection (Section 2.1.1) and edge aggregation-based segmentation (Section 2.1.2). Based on the detection results, iExplorerGame then assigns a feature vector for each predicted widget that takes the widget category into account, and ranks the widgets according to the priority computed with the feature vectors. iExplorerGame finally conducts the operation (click or slide) on the widget with the highest priority.

2.1 Mixed GUI Widget Detection for Mini-Games

To effectively detect GUI widgets for mini-games, we propose a mixed method that combines deep learning-based object detection (DLOD) and traditional edge aggregation-based segmentation (EAS) (e.g., canny edges). We utilize the combination to detect GUI widgets according to their high-level semantic feature (identified by DLOD) and low-level texture feature (identified by EAS).

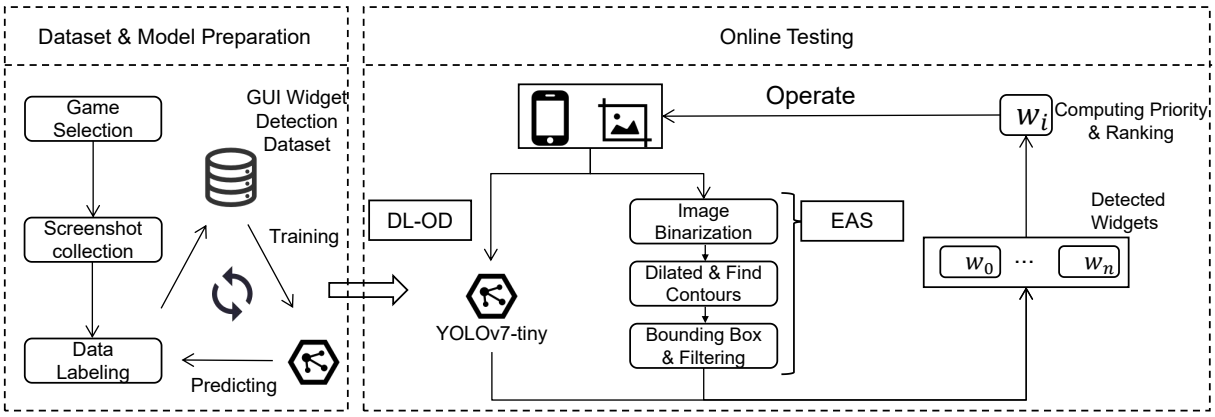

Figure 2: Overview of iExplorerGame.

2.1.1 Deep Learning-Based Object Detection. To better capture the patterns of diverse GUI widgets in mini-games, we propose the first GUI widget detection dataset for mini-games, which involves 343 mini-games and covers five GUI widget categories. The prepared dataset is then employed to train the GUI widget detection model Compared with the existing GUI widget detection benchmark [67] on mobile games which only considers clickable widgets (e.g., buttons), our dataset is more diverse in GUI design by involving different widget categories such as sliding widgets . We first introduce the covered five GUI widget categories, and then elaborate on the dataset preparation process and the deep learning-based object detection model, respectively, in the following.

Widget Category Design. According to prior work [67] and our observation on mini-games, we classify GUI widgets into five categories including *click*, *slide*, *target*, *popup*, and *checkbox*. We illustrate a few examples for each category in Figure 3, and introduces the details as below.

(1) **Click** category indicates clickable widgets such as buttons as shown in Figure 3 (a). Such category has been commonly considered by previous image-based detection approaches [63, 67].
(2) **Target** is the kind of widgets that play a guiding role and usually occur in the novice guidance period as Figure 3 (b) shows. The *target* widget appears in the form of a directional sign such as a hand and an arrow. By detecting the widget, iExplorerGame can efficiently conduct the required operation to pass through the pages for novice guidance.
(3) **Slide** category represents the widgets that can trigger actions via sliding. Detecting slidable widgets can facilitate us in conducting sliding operations and triggering more actions.
(4) **Popup** category is a kind of windows that display important information. Detecting the bounding box of popups can help to efficiently close them (e.g., the close button usually locates at the upper right corner of popups).
(5) **Checkbox** category is designed for the unchecked boxes in games. We propose to detect checkboxes to handle the situation that only if the checkbox is checked, the game can be further played (e.g., checking user privacy items).

Figure 3: Examples of five categories of widgets.

Dataset Preparation. Our dataset preparation process consists of three main steps: game selection, screenshot collection, and data labeling. The details of each step are as follows:

(1) **Game selection.** To select the representative mini-games, we first select the top 100 popular ones. In addition to the popularity, we also consider GUI design styles (including color palette, iconography, and textures) as the other aspect of selection. Therefore, we manually involve other 243 mini-games ensuring that they represent a variety of styles.
(2) **Screenshot collection.** We select game screenshots from the mini-game testing platform of WeChat that records the log and screenshots of each testing process. To improve the diversity of selected screenshots, our screenshot selection criteria are 1) including launching pages and main pages, 2) including all the pages about novice guidance, and 3) other non-repeating pages. We collect 6,024 screenshots for the 343 selected mini-games in total.
(3) **Data labeling.** To diminish the manual cost of labeling, the first two authors label 2,000 screenshots via the image labeling tool LabelImg[1]. The first round labeling lasts one week. Based on the labeled data, we train a preliminary object detection model based on YOLOv7 and use it to label the next batch of screenshots (batch size 1,000). With the results predicted by the model, we manually check and rectify the

[1] https://github.com/heartexlabs/labelImg

labels. The process of rectifying a batch of screenshots requires a full day's worth of time. We repeat this training, prediction, and manually labeling process multiple times until all the screenshots are labeled. After data labeling, all of the screenshots and their labels are double-checked by the first two authors. In summary, the statistics of our dataset are shown in Table 1.

GUI Widget Detection. To detect GUI widgets for mini-games, we utilize YOLOv7 [60], a state-of-the-art object detection model, to train on our labeled dataset. We only introduce the model's input and output here, and readers can refer to the original paper for more details.

Given the input screenshot s of a game page, YOLOv7 can predict the GUI widgets on the page. Specifically, YOLOv7 predicts the bounding box (i.e., minimum circumscribed rectangle), category, and confidence score of each GUI widget:

$$\begin{aligned} &B, C, Conf = D(s) \\ &where \quad B = \{(x_i, y_i, h_i, w_i)\}_{i=1}^{n}, \end{aligned} \tag{1}$$

where n denotes the number of widgets predicted by the detector and B is the bounding box set. Each bounding box is a rectangle, and (x_i, y_i, h_i, w_i) indicates the center coordinate, the height, and the width of the bounding box, respectively. C and $Conf$ denote the widget category set and corresponding confidence score set, respectively.

2.1.2 Edge Aggregation-Based Segmentation. Although the DLOD approach can detect many widgets, it is still plagued by over-fitting and data-drifting problems [18, 32, 41]. Section 1 highlights the diverse GUI designs of mini-games, posing a challenge to accurately detect different widgets. Therefore, to enhance the performance of iExplorerGame in detecting rare widgets, we propose to incorporate a traditional edge aggregation-based segmentation (EAS) approach.

Motivated by our observation that GUI widgets generally have clear edges around them, we propose to detect potential GUI widgets based on edge features (i.e., Canny [19]). First, we use Canny to detect the edges in the input screenshot and output a binary image with all detected edges. Second, to aggregate the adjacent edges, we conduct an "opening operation" to the binary edge image:

$$Open(b_s) = Dilation(Erosion(b_s)) \tag{2}$$

where b_s denotes the binary screenshot processed by Canny. Functions $Dilation$ and $Erosion$ represent the image dilation and erosion.

Third, we compute a bounding box for each connected domain in the dilated image via the *BoundingRect* function in OpenCV [17]. Fourth, we filter the computed bounding boxes based on the rules listed in Table 2 following [46] and output the final prediction.

The edge aggregation based segmentation approach is capable of detecting all the GUI widgets that have clear edges around (e.g., text and buttons) and has a very high recall rate. Thus, this approach can complement DLOD, ensuring the inclusion of all widgets with clear edges and preventing omissions.

2.2 Category-Aware Testing Strategy

In this section, we propose a category-aware testing strategy to explore the mini-games based on the detected GUI widgets. The strategy includes two aspects, i.e., the priority-based clicking strategy and sliding strategy.

2.2.1 Priority-Based Clicking Strategy. To decide which detected widgets to click, we propose the priority-based clicking strategy to automatically determine the widget to click. For our clicking strategy, we take all widgets predicted by the EAS approach and "*click*" widgets predicted by the DLOD approach into account. For each widget of a page screenshot, we assign a feature vector with nine dimensions to calculate its priority:

$$\boldsymbol{V} = \{v_1, v_2 ... v_9\}, \quad p = \sum_{i=1}^{9} w_i * v_i \tag{3}$$

where $\boldsymbol{V}$ is the feature vector and v_i denotes the value of i-th dimension. w_i indicates the weight assigned to each dimension to calculate the widget priority p. In this paper, the weights are set empirically, where w_1 and w_4 are negative and the others are set positive.

Among the nine dimensions in the feature vector, dimensions (1) to (5) are method-agnostic (i.e., they are only related to the widgets' location and brightness, which can be applicable to the widgets predicted by both DLOD and EAS approaches). For dimensions (6) to (9), these scores are method-specific, i.e., computed based on the DLOD results.

In this section, we will introduce each dimension of the feature vector.

(1) **Heat Score** v_1 . We initialize a zero heat score to each detected *click* widget on a page. Each click operation will accumulate the heat score for the corresponding widget. We set the weight of this dimension $w_1 < 0$ because we expect the iExplorerGame to conduct diverse operations when visiting the same page multiple times. Note that in this work, we make the assumption that identical pages yield similar results in terms of GUI widget detection. Leveraging this assumption, we identify pages that have been previously explored, and consequently, we can know whether a detected widget has been operated.

(2) **Triggerable Score** v_2 . We also initialize a triggerable score as zero for each widget on a page. If the page changes through an operation, the operation score s_o of the corresponding widget will be accumulated. Then the triggerable score can be computed as follows:

$$v_2 = \text{sigmoid}(s_o - 5) = \frac{1}{1 + e^{-s_o+5}}. \tag{4}$$

This score is designed to give more priority to the widgets that can trigger page switches. The reason we use sigmoid is that we expect to prioritize predicted widgets that are validated to be triggerable, by increasing their scores with each triggering occurrence. Simultaneously, we anticipate that iExplorerGame will explore additional widgets, rather than trapping itself in those that result in page changes.

(3) **Potential Return Button Score** v_3 . If a widget is located in the upper left part of the screenshot ($rate_x < 0.25$ and $rate_y < 0.25$, where $rate$ denotes the relative position of the axis and the left upper is the origin), and the aspect ratio of its bounding box is between 0.25 and 2 (reasonable rectangle), it will be considered as a potential return button [14, 31] (the score is set to 1 otherwise 0). The purpose of the feature dimension is to give more priority

Table 1: Statistics of widget detection dataset. Each widget has exactly one label.

#Mini-Games	#Screenshots	#Click	#Target	#Slide	#Popup	#Checkbox	All
343	6,024	57,073	853	2,913	2,501	306	63,676

Table 2: GUI widget filtering rules. B is the set of bounding boxes predicted by the object detection approach and edge aggregation-based approach, and b is a bounding box under consideration ($b \in B$). Each bounding box has the attribute height ($b.h$), width ($h.w$), and area ($h.area$), respectively. H and W indicate the height and width of the screenshot, respectively. The functions $vDis$ and $hDis$ measure the vertical and horizontal distance of the centers of two bounding boxes, respectively

ID	Predicate	Rule
R1	isTooSmall(b)	$b.w < 15$ **or** $b.h < 15$
R2	isTooBig(b)	$b.w > 0.8W$ **or** $b.h > 0.8H$ **or** $b.area > 0.8W * H$
R3	isTooSlimOrFat(b)	$b.w/b.h < 10$ **or** $b.h/b.w < 10$
R4	isDuplicate(b, B)	$\exists b' \in B : (b' \cap b).area > 0.8 * b'.area$ **and** $vDis(b', b) + hDis(b', b) < 30$

to the button that returns to the main page, preventing the testing process stuck on one page.

(4) **Overlapping Score** v_4 . This dimension denotes the number of widgets that overlap with the current widget. We design this dimension based on the intuition that an isolated widget is more likely to be a button.

(5) **Brightness Score** v_5 . In game GUI design [31], clickable buttons are generally brighter than un-clickable ones. Here, the brightness score is calculated by taking the average RGB value of the pixels in the widget area.

(6) **Prediction Source Score** v_6 . This dimension denotes whether the result is predicted by DLOD (score is 1) or EAS (score is 0). Considering that the DL-based object detection approach has much higher precision (as discussed in Section 4.1), we give higher priority to its detection results.

(7) **Intention Score** v_7 . This score measures the operation intention of the widget. We observe that if a "*target*" widget exists on a page (novice guidance), the widgets closer to the *target* are more likely to be intended to be operated. Thus, the intention score of a widget is computed as follows (if any "*target*" widget is detected):

$$v_7 = \text{sigmoid}(-d_t) = \frac{1}{1 + e^{d_t}}, \tag{5}$$

where d_t denotes the distance from the widget to the "*target*" widget (if multiple *target*s are detected, select the one that has the highest confidence score).

(8) **Checkbox Score** v_8 . As introduced in Section 2.1.1, *checkbox* is also the kind of widget that we need to give high priority. Thus, this score is set to 1 when the widget is predicted as a *checkbox* otherwise 0.

(9) **Potential Close Button Score** v_9 . If a *click* widget is located in the upper right part of a *popup* widget ($rate_x > 0.6$ and $rate_y < 0.4$) and similar to a square ($h : w \in [0.9, 1.1]$), we will regard it as a potential close button (the score is set to 1 otherwise 0). We design this dimension to close the popup rapidly.

For method-specific dimensions v_6 to v_9 related to the DLOD prediction, we multiply a confidence score for computing the priority (i.e., $Conf$ in Equation 1). We observe that the widgets predicted with higher confidence are more likely to be correct.

Based on the priority computed for each widget on the screenshot, we select the widget with the highest priority to operate. Specifically, we will randomly click a location inside the bounding box of the corresponding widget.

2.2.2 Sliding Strategy. As aforementioned that sliding operations in games have a higher requirement for precision, i.e., requiring specific start or end points. Therefore, in this work, we propose two strategies for sliding.

Based on *slide* widget. Similar to the *click*, *slide* widgets also have operation feature vectors. If a *slide* widget with the highest priority is selected to operate, we will randomly slide it in a random direction with a random distance.

Based on the movement of *target* widget. In the novice guidance period, some games require users to slide a widget from one place to another. Specifically, game developers utilize a hand or an arrow (we detect as *target*) to move between two points and conduct guidance for sliding. Therefore, if a *target* is detected moving on a page, we will record the *target*'s track and attempt to slide along it.

3 EXPERIMENT SETUP

3.1 Research Questions

To study the effectiveness of our iExplorerGame, we investigate the following four research questions:

- **RQ1:** How effective is iExplorerGame in detecting GUI widgets for mini-games?
- **RQ2:** How capable is our iExplorerGame to test mini-games?
- **RQ3:** What is the impact of each part in iExplorerGame on the performance of mini-game testing?
- **RQ4:** How effective is iExplorerGame in handling complex game pages?

We design RQ1 to explore the effectiveness of our proposed approaches in detecting GUI widgets for mini-games. We study the effectiveness from two aspects including the performance on seen games and unseen games, respectively. RQ2 aims to verify whether our iExplorerGame can effectively explore mini-games. In RQ3, we study the impact of each part in iExplorerGame, including EAS, DLOD, and multi-category design. RQ4 provides a case study to

Table 3: Number of games of different game types.

	Puzzle	Card	Match-3	RTS	RPG	SLG	Fighting	Other	All
#Games	88	69	26	15	34	37	27	23	319

Algorithm 1: Algorithm of computing number of aggregated pages.

Input: Screenshot set S.
Output: Number of aggregated pages NAP.

```
1  Initialize screenshot memory M=[]
2  Initialize NAP=0
3  for s in S do
4      Initialize SIM=[]
5      for m in M do
6          SIM.append(sim(s, m))
7      end
8      if min(SIM)<T then
9          NAP++
10     end
11     M.append(s)
12 end
13 return NAP
```

further investigate the effectiveness of iExplorerGame, especially for the complex game pages.

3.2 Evaluation Metrics

In this section, we will introduce the evaluation metrics used for evaluating the effectiveness of GUI widgets detection and game testing, respectively.

3.2.1 Metrics of GUI Widgets Detection. We first introduce the basic concepts of the metrics used in the object detection area. Generally, an Intersection over Union (IoU) threshold K is set to determine the true positives. If the proportion of intersection and union area between the predicted bounding box and the ground truth box is larger than K, the prediction is regarded as a true positive (TP). Otherwise, the predicted box is a false positive (FP). If a bounding box in ground truth is not detected, it will be a false negative (FN).

Precision and **Recall** measure the proportion of correctly predicted GUI widgets among model prediction and the capability of model detecting as many GUI widgets as possible in the screenshots, respectively.

$$Precision = \frac{TP}{TP+FP}, Recall = \frac{TP}{TP+FN} \tag{6}$$

where TP, FP, and FN denote the number of true positives, false positives, and false negatives respectively.

Mean Average Precision@K (MAP@K) is another commonly used metric in the object detection area [25, 50, 60]. Given the model prediction on a screenshot, sort the predicted BBox in the descending order of $Conf$ and draw the Precision-Recall curve (the true positives are computed based on IoU threshold K). The average precision (AP) is computed as follows:

$$AP = \int_0^1 p(r)dr \tag{7}$$

where r denotes the recall and $p(r)$ indicates the corresponding precision of r in the P-R curve. According to APs on the testing screenshots, the MAP is defined as the mean of all APs.

Table 4: Comparison of our EAS, DLOD, and previous GUI widget detection methods.

Methods	Precision	Recall	MAP@50	MAP@95
UIED [63]	0.243	0.355	0.279	0.141
Yan et al.[67]	0.530	0.481	0.448	0.258
EAS	0.202	**0.975**	0.234	0.170
DLOD	**0.790**	0.826	**0.784**	**0.703**

3.2.2 Metrics of Game Testing. **Number of Aggregated Pages (NAP)** measures the non-repeating pages that a testing process explores in the game, which is also a long-used evaluation metric in WeChat mini-game testing platform.

During testing, we will record the screenshot after each operation as the screenshot set S. The algorithm of computing NAP is depicted in Algorithm 1. The algorithm works by comparing each screenshot to the ones stored in memory. If the similarity between the screenshots is less than a threshold T (set as 0.5 in our work), they are considered to belong to the same aggregated page, and NAP is incremented. The similarity function used in line 6 is implemented by comparing the correlation score between the pixel histograms of two input images.

3.3 Implementation Detail

3.3.1 Deep Learning Based Object Detection Model Training. We randomly select the screenshots of 20 games with 373 screenshots as the test set (unseen games) and randomly split the remaining as the ratio of 9:1 to construct the training set and validation set (seen games).

To detect GUI widgets efficiently, we choose YOLOv7-tiny, a lightweight version of the YOLOv7 series models, as our GUI widgets detection model. All of the training is run on an NVIDIA T4 with 16GB of graphic memory. We use the default hyperparameter setting of YOLOv7 provided in the official repository[2]. We train the model for 250 epochs and select the checkpoint that performs the best on the validation set.

After obtaining a trained model, we use Open Neural Network Exchange (ONNX) [3] to export the model to further improve the efficiency of the model running on CPU environments.

[2] https://github.com/WongKinYiu/yolov7

Table 5: Widget detection results on seen games and unseen games.

Classes	Seen Games				Unseen Games			
	Precision	Recall	MAP@50	MAP@95	Precision	Recall	MAP@50	MAP@95
Click	0.924	0.882	0.937	0.757	0.790	0.826	0.784	0.703
Target	0.892	0.784	0.807	0.484	0.779	0.778	0.769	0.510
Slide	0.931	0.692	0.716	0.631	0.812	0.749	0.681	0.613
Popup	0.926	0.943	0.969	0.871	0.930	0.891	0.934	0.839
Checkbox	0.984	0.864	0.980	0.519	0.907	0.724	0.824	0.425
Average	0.916	0.773	0.822	0.591	0.824	0.774	0.726	0.535

3.3.2 Online Testing Environment. We select the top 100 popular mini-games according to WeChat mini-game monthly ranking and randomly sample 219 ones that *do not overlap* with the labeled games as our testing games. We divide the mini-games into 8 categories (including puzzle games, card games, match-3 games, real-time strategy games (RTS), role-playing games (RPGs), simulation games (SLG), fighting games, and other types) and the statistics are shown in Table 3.

We run all the experiments in the WeChat mini-game online testing platform, and each testing process is run on the docker environment that is allocated with two GB of memory and three cores of CPU. Each screenshot is resized into 640×640 pixels and fed into the GUI widget detection model. We run each testing approach on ten randomly selected smartphone devices (including eight Android devices and two IOS devices) for ten minutes. The time interval between two operations is fixed as one second. The weights we use to calculate the priority for the dimensions (v_1 to v_9) in the feature vector are -30, 45, 30, -10, 0.2, 100, 400, 200, and 30, respectively.

4 EXPERIMENT RESULTS

4.1 RQ1: Effectiveness of GUI Widget Detection

4.1.1 Comparison with Other GUI Widget Detection Approaches. In this section, we compare our proposed DLOD and EAS approaches with the existing GUI widget detection approaches including UIED [63] and the method used by Yan et al. [67]. For UIED, we use the official repository [3] to detect GUI widgets. In addition, we employ the benchmark provided by Yan et al. to train the identical object detection model (YOLOv7-tiny) and then test on our dedicated test set. Due to the limited support of the previous techniques and EAS approach in terms of category information about detected widgets, we evaluate their effectiveness on the *click* categories in our dataset. The experiment results as shown in Table 4. From the table, it is evident that UIED exhibits suboptimal performance across all four evaluation metrics. This suggests that UIED has limited capabilities in accurately detecting GUI widgets within mini-game scenaiors. This underperformance can largely be attributed to the considerable disparity between the GUI designs of mobile apps and mini-games. For the model trained by the benchmark proposed by Yan et al. [67] (line 2), we can find an obvious improvement compared with UIED. However, the model can only retrieve less than half of the clickable buttons (0.48 recall) and make inaccurate predictions (0.53 precision). We suppose this unsatisfactory performance is led by that only nine games with 2,961 screenshots are involved in this benchmark, resulting in a worse generalization.

For EAS, we can find that its precision and MAPs are not satisfying (i.e., about 0.2). However, it achieves the highest recall (0.975) among all of the four approaches. This is attributable to the edge aggregation technique, which identifies all GUI widgets with clear edges, covering a broad spectrum of the game page. Our DLOD performs well on all four metrics; however, its recall rate is limited to 0.826, which may result in the failure to detect some essential widgets. As a result, we propose a mixed approach that combines EAS and DLOD to achieve both precision and recall when detecting GUI widgets.

4.1.2 Performance of Detecting Different Category of GUI Widgets for Mini-Games. This section presents a detailed investigation into the effectiveness of the DLOD approach for detecting various categories of game GUI widgets in both seen and unseen games. The detection results are described in Table 5.

Overall, the results of our widget detection model present a high level of performance. For the detection performance on seen games, the DLOD achieves great performance (i.e., the average precision is more than 0.9 and MAP@50 is over 0.82). The results demonstrate that the model can learn the widget features and detect widgets in the seen games well. For detection results on unseen games, the model shows a performance degradation on the metrics compared with the results in seen games. The model still achieves precision at 0.82 and recall at 0.77. The performance indicates that the model trained on large-scale game screenshots has a great generalization ability to the games not involved in the training set.

Comparing the performance among different categories, we find that the *popup* category obtains the best detection performance, achieving over 0.8 MAP@95 in seen and unseen games. This is because the characteristics of popups are the most obvious, i.e., large and usually located in the middle of the screen. The second best-detected widget category is *click*. Despite the high diversity in clickable widget design within games, the use of a large amount of training data can effectively address this issue and enable the model to accurately predict the *click* widgets. Furthermore, we can observe that the model achieves a remarkable Mean Average Precision (MAP) score of 0.98 for the *checkbox* category on seen games. We speculate that the prominent detection accuracy is due to the distinctive feature of the *checkbox* category, which includes a circle or a box in proximity to text. However, the MAP@95 of the category drops obviously compared with MAP@50. This may

[3] https://github.com/MulongXie/UIED

Table 6: Average number of aggregated pages (NAP) explored by our iExplorerGame of different game types. DLOD (Click) denotes we use DLOD to predict only *click* widgets.

NAP	Puzzle	Card	Match-3	RTS	RPG	SLG	Fighting	Other	Overall
Random	24.93	25.74	20.69	25.53	19.74	21.49	22.41	20.52	23.30
EAS	32.40	34.32	27.65	33.93	26.76	29.30	29.63	27.35	30.94
DLOD (Click)	35.39	40.87	31.73	36.60	27.68	27.32	28.07	30.57	33.61
EAS + DLOD (Click)	35.66	44.68	32.31	35.53	31.12	32.11	30.33	30.09	35.58
EAS + DLOD (Multi) (iExplorerGame)	**44.23**	**52.93**	**39.12**	**46.53**	**38.00**	**39.43**	**44.81**	**37.04**	**44.11**

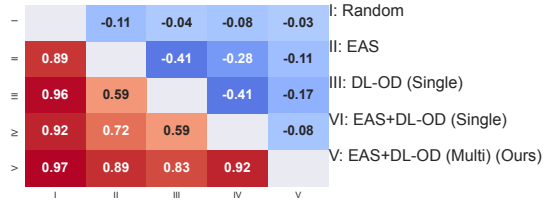

Figure 4: Breakdown analysis of iExplorerGame. The numerical value within the unit indicates the proportion of evaluated games in which the horizontal approach outperformed the vertical approach with respect to NAP.

be attributed to that the bounding boxes of *checkbox* widgets are relatively small (i.e., no more than 30 pixels), and the model is hard to predict their locations perfectly. For the *target* category, due to its limited training instances and various appearances, a relatively lower MAP is observed. *Slide* widgets appear close to clickable ones, thus it is hard for models to detect the category in unseen pages (i.e., achieving the lowest recall and MAP@50).

4.2 RQ2: Effectiveness of Mini-Game Testing

We report the effectiveness of our iExplorerGame in testing mini-games in Table 6. From the table, we can find that our iExplorerGame achieves promising NAPs among all game types. Overall, iExplorerGame is able to explore more than 44 non-repetitive pages in mini-game testing. To better evaluate the effectiveness of iExplorerGame, we compared it to our previous mini-game testing approach that clicks on a random location of pages (the first row). The table presents that iExplorerGame improves the NAP by 77% ∼ 106% on different game types, and the average NAP improvement is 89%. Furthermore, we conduct the breakdown analysis in Figure 4 that presents the pair-wise comparison results. In Figure 4 we can find that iExplorerGame explores more pages in 97% of tested mini-games (309/319) compared with Random. The results demonstrate that our proposed iExplorerGame is effective in mini-game testing.

In addition, we also compare our iExplorerGame with human testers. We enlisted ten software testing professionals and tasked them with playing the 30 most popular mini-games for a duration of 10 minutes. We ask the professionals to explore as many new pages in the game as possible. The results are shown in Figure 5. Overall, iExplorerGame explores an average of 43 pages among 30 tested mini-games, which achieves 63% performance of software testing professionals (i.e., 68). Notably, iExplorerGame outperforms the professionals in seven mini-games, underscores its effectiveness in mini-game exploration. We further examined the seven cases where the NAP of iExplorerGame is less than half of that of professionals, such as games 15 and 16. Our investigation revealed that iExplorerGame's poor performance was primarily attributed to the mini-games that demanded a holistic comprehension of the gameplay, such as achieving a specific score to progress to the next level.

Additionally, to evaluate the capability of iExplorerGame in finding bugs for mini-games, we deploy iExplorerGame in our WeChat mini-game testing platform. In the past 30 days, iExplorerGame has tested more than 76,000 mini-games and successfully trigger 22,144 bugs. These bugs have been confirmed and feedback to the game developers. Such results show that iExplorerGame can find bugs for mini-games in industrial scenarios.

4.3 RQ3: Ablation Study

To evaluate the effectiveness of each component of iExplorerGame, we conduct an ablation study and report the results in Table 6 and Figure 4. The used ablation baselines are 1) **Random**: we randomly click a position in the screenshot, 2) **EAS**: we click the widget with the highest priority predicted by EAS (through model-agnostic dimensions v_1 to v_5), 3) **DLOD (Click)**: we only consider *click* categories among DLOD's detection results and choose the widget with the top priority to click, 4) **EAS +DLOD (Click)**, we compute priority based on detection results of EAS and DLOD (Click) and additionally use the dimension v_6 to distinguish the prediction from the two approaches, and 5) **EAS + DLOD (Multi)**: we take all of the widget detection results into account and use all the nine dimensions of the feature vector.

Comparing the first two rows, we observe that using edge aggregation based segmentation with the method-agnostic feature vector significantly improves the metric NAP by 33% compared with the approach that randomly clicks on a position. In terms of breakdown analysis, EAS also explores more pages in 89% games under testing. This improvement is attributed to the fact that EAS significantly reduces the search space to less than a hundred potential widgets, as opposed to the number of pixels.

Moreover, we use the DLOD approach to detect *click* widgets (marked as DLOD (Click)) in the third row. We observe that DLOD (Click) outperforms EAS by 8%, indicating the benefit of high precision of widget detection in game testing. However, we find eight bad cases when conducting further analysis, where DLOD (Click) only explores less than ten pages. After incorporating the detection results we come to the conclusion that DLOD (Click) fails to detect some critical widgets (e.g., the close button on a popup) and results in being stuck on a page permanently.

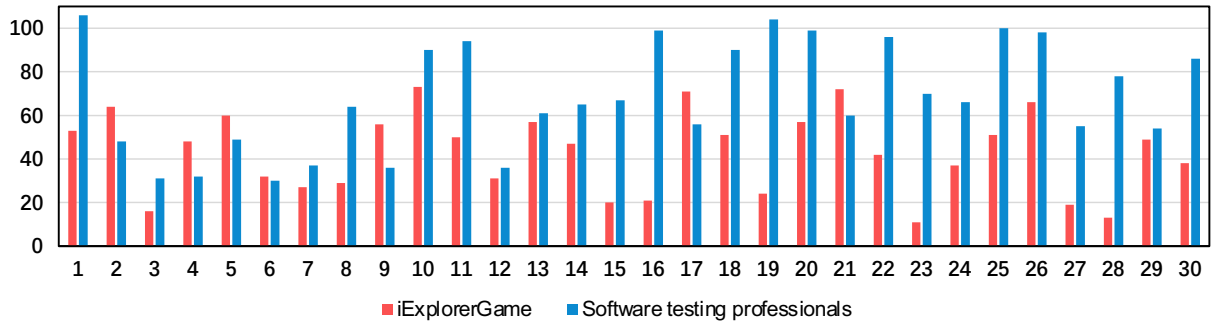

Figure 5: Comparison of iExplorerGame with software testing professionals. The x-axis denotes the game index (1 to 30) and the y-axis indicates NAP.

Figure 6: Examples of GUI widget detection results for clicking operations. The text and number on the DLOD detection results denote the widget category and confidence score, respectively.

In the fourth row, we combine the detection results of DLOD (click) and EAS, and additionally use the dimension v_6 to distinguish the prediction from the two approaches. Compared with singly using EAS or DLOD (click), the combination outperforms the two approaches by 15% and 6%, respectively. The results demonstrate that the combination of EAS and DLOD is able to achieve higher NAP and prevent failing to retrieve critical widgets on the pages. Our proposed iExplorerGame (the fifth row) shows the best capacity of game testing, i.e., outperforming the second-best approach by 24% in terms of average NAP and exploring more pages in 92% mini-games. The results indicate that our multi-category design in mixed GUI widget detection combined with our testing strategy is effective.

4.4 RQ4: Case Study

4.4.1 Clicking Operation Examples. This section selects six screenshots from different games and shows the detection results of our proposed EAS and DLOD, respectively.

As examples in Figure 6 show, the GUI design of different games varies greatly. For the screenshots of relatively simple pages (Figure 6 (a) to (c)), EAS is capable to detect the isolated widgets with clear edges. However, when the page becomes complex, EAS suffers from low precision. For instance, in the page shown in Figure 6 (e), EAS predicts tens of widgets on the girl but actually, there is no clickable widget. For the page in Figure 6 (f), the buttons on the bottom of the screenshot have a similar color to the background that EAS fails to detect. Furthermore, a hand exists on the screenshots in Figure 6 (a) and (f) to prompt users the place to click. However, EAS cannot distinguish it from detected widgets, which may waste time on clicking the un-triggerable buttons.

Furthermore, we observe that DLOD detects GUI widgets more precisely. In Figure 6 (a), EAS wrongly detects un-triggerable widgets (e.g., texts) that DLOD ignores. Observing Figure 6 (b) and (c), DLOD also makes more precise widget predictions. In addition, in the examples shown in Figure 6 (d), a popup suspends in front of the main page and the background is dark, meaning that the widgets on the background are inoperable. Although EAS successfully detects

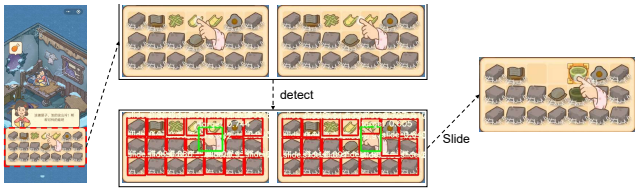

Figure 7: Examples of GUI widget detection results for sliding operations. Green and red boxes indicate the *target* and *slide* widgets, respectively.

the widgets on the popup, it also detects the un-triggerable widgets. However, DLOD accurately predicts the widgets on the popup. For the complex pages in Figure 6 (e) and (f), DLOD is also able to detect all of the buttons. In addition, for the pages that belong to novice guidance (Figure 6 (a) and (f)), DLOD can detect the *target* widgets (in the green boxes) and we prioritize clicking the widgets close to them.

4.4.2 Sliding Operation Examples. Figure 7 shows an example of how our iExplorerGame conducts sliding operations in the novice guidance scenarios.

In the game screenshot, iExplorerGame detects all of the slidable widgets into red bounding boxes. However, simply sliding one of them cannot pass through the novice guidance. As shown in the figure, a hand moves between two identical items, guiding users to slide one item to the other and merge them into a new item. Our iExplorerGame identifies the hand as the *target* category, as shown in the green box. Then the iExplorerGame captures the movement trajectory of the hand and guides it along this path, thereby successfully achieving the intended guidance and synthesizing a new item.

5 THREATS TO VALIDITY

One threat to validity lies in the low computation resource we can utilize to conduct GUI widget detection. As introduced in Section 3.3, we run the testing script in the docker environment with 3 CPU cores and 2 GB memory. In addition, game testing is also a time-sensitive task. The available resource limits us from utilizing powerful GUI widget detection models. To solve the threat, we use YOLOv7-tiny with only 6.7 million parameters to detect GUI widgets, aiming to achieve a balance between efficiency and effectiveness.

Another threat is the weights we use for computing priority for detected widgets. In this work, we empirically set the weights which may not be optimal. This is because of the huge cost of evaluating different weight settings and limited computational resources to utilize weights tuning techniques such as reinforcement learning.

6 RELATED WORK

6.1 Automated GUI Testing

There have been a series of work focusing on automated GUI testing. The pioneering work is Monkey [6], which simply conducts random actions in Android apps according to pre-defined probability distributions. It is efficient and effective, even outperforming numerous research tools [61, 69]. Other tools [39, 52, 54, 66] randomly generate input for Android apps to reveal potential crashes. Model-based testing approaches [8, 9, 11, 12, 22, 28, 33, 53, 64, 65] build models for apps with dynamic or static strategies to describe apps' behaviors, deriving test cases from built models and achieving performance. Combined with symbolic execution and evolutionary algorithms, the work [10, 24, 40, 42] conduct systematic strategies to generate delicate inputs to improve code coverage. With the development of machine learning (ML), numerous ML-based approaches have been proposed. The work [16, 30, 34, 35] train deep learning models to learn the testing process which can be generalized to new apps. Other work [7, 45, 57, 58, 70] also leverage reinforcement learning to train the model through the testing process. Recently, Liu et al. [37] utilize large language models to generate input for apps.

6.2 GUI Widget Detection

Previous work have attempted to extract GUI widgets on the app screenshots. In early work, researchers utilize edge features in app screenshots to detect potential GUI regions [43, 44], or detect specific GUI widgets based on template matching [13, 23, 47, 68]. Recently, some work [20, 62] use object detection techniques to detect GUI widgets in mobile apps. In addition, authors in the work [21, 63] combine image segmentation approaches and deep learning techniques to detect GUI widgets. However, these techniques are proposed for GUI widget detection for mobile apps, where the GUI design is far different from that of mini-games and thus is not applicable in our scenarios.

Ye et al. [67] conduct an empirical study of GUI widget detection for industrial mobile games and propose a benchmark. However, the benchmark only includes nine mobile games and considers the widgets for click operation (i.e., buttons). Consequently, models trained on this benchmark may struggle to recognize more complex widgets and operations in large-scale mini-game testing.

6.3 Object Detection

Object detection approaches can be briefly classified into two categories including two-stage [18, 26, 27, 29, 51] and one-stage [15, 25, 36, 48, 50, 55, 56, 59, 60] ones. For two-stage approaches such as RCNN series work [27, 51], they first conduct region proposal and then utilize a classifier to predict a label for each region. Two-stage approaches achieve state-of-the-art performance but also bring additional time consumption. Two improve the detection efficiency, Redmon et al. propose YOLO [48], a unified object detection framework that can predict bounding boxes and labels in one evaluation. Up till now, there have been a series of work being proposed based on YOLO including YOLOv2 [49], YOLOv3 [50], YOLOv4 [15, 59], YOLOX [25], and YOLOv7 [60]. Recently, some other one-stage object detection approaches have been proposed. For instance, Tian et al. [55] utilize fully convolutional networks [38] for anchor-free object detection.

7 CONCLUSION

In this paper, we have proposed iExplorerGame, a unified framework for mini-game testing. Specifically, we have introduced mixed GUI widget detection approach to detect GUI widget for mini-games, and then a category-aware testing strategy to effectively explore mini-games. The iExplorerGame has been deployed in WeChat mini-game testing platform, and found 22,144 real bugs in the past 30 days.

REFERENCES

[1] 2022. Data.ai: Global mobile game revenue dipped 5% to $110bn in 2022. https://www.gamesindustry.biz/dataai-global-mobile-game-revenue-dipped-5-to-110bn-in-2022.

[2] 2022. LayaAir. https://layaair.layabox.com/.

[3] 2022. ONNX. https://onnx.ai/.

[4] 2022. Unity. https://unity.com/.

[5] 2022. WeChat Mini-Game. https://developers.weixin.qq.com/minigame/introduction.

[6] 2023. Android Monkey. https://developer.android.com/studio/test/monkey.

[7] 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.

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

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

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

[11] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.

[12] 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.

[13] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. 2015. scvRipper: video scraping tool for modeling developers' behavior using interaction data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 673–676.

[14] Jessa Bekker and Jesse Davis. 2020. Learning from positive and unlabeled data: A survey. *Machine Learning* 109 (2020), 719–760.

[15] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934* (2020).

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

[17] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[18] Zhaowei Cai and Nuno Vasconcelos. 2018. Cascade r-cnn: Delving into high quality object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 6154–6162.

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

[20] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.

[21] 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.

[22] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. 31–36.

[23] Morgan Dixon and James Fogarty. 2010. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1525–1534.

[24] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 419–429.

[25] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. 2021. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430* (2021).

[26] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.

[27] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587.

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

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE transactions on pattern analysis and machine intelligence* 37, 9 (2015), 1904–1916.

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

[31] Raph Koster. 2013. *Theory of fun for game design.* " O'Reilly Media, Inc.".

[32] Bartosz Krawczyk, Bernhard Pfahringer, and Michał Woźniak. 2018. Combining active learning with concept drift detection for data stream mining. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2239–2244.

[33] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 115–127.

[34] 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.

[35] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.

[36] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 21–37.

[37] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. *arXiv preprint arXiv:2212.04732* (2022).

[38] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3431–3440.

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

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

[41] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. 2022. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems* 4 (2022), 77–94.

[42] 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.

[43] 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.

[44] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 248–259.

[45] 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.

[46] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2020. GUI-guided test script repair for mobile apps. *IEEE Transactions on Software Engineering* 48, 3 (2020), 910–929.

[47] 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.

[48] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[49] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7263–7271.

[50] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[51] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015).

[52] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. 1–5.

[53] 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.

[54] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 204–215.

[55] Zhi Tian, Chunhua Shen, Hao Chen, and Tong He. 2019. Fcos: Fully convolutional one-stage object detection. In *Proceedings of the IEEE/CVF international conference on computer vision*. 9627–9636.

[56] Zhi Tian, Chunhua Shen, Hao Chen, and Tong He. 2020. Fcos: A simple and strong anchor-free object detector. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 4 (2020), 1922–1933.

[57] Thi Anh Tuyet Vuong and Shingo Takada. 2018. A reinforcement learning based approach to automated testing of android applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 31–37.

[58] Thi Anh Tuyet Vuong and Shingo Takada. 2019. Semantic Analysis for Deep Q-Network in Android GUI Testing.. In *SEKE*. 123–170.

[59] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2021. Scaled-yolov4: Scaling cross stage partial network. In *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*. 13029–13038.

[60] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2022. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696* (2022).

[61] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 738–748.

[62] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 307–317.

[63] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: a hybrid tool for GUI element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1655–1659.

[64] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. 2017. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 42–53.

[65] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 16*. Springer, 250–265.

[66] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. 68–74.

[67] Jiaming Ye, Ke Chen, Xiaofei Xie, Lei Ma, Ruochen Huang, Yingfeng Chen, Yinxing Xue, and Jianjun Zhao. 2021. An empirical study of GUI widget detection for industrial mobile games. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1427–1437.

[68] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. 2009. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. 183–192.

[69] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: are we really there yet in an industrial case?. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 987–992.

[70] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 772–784.

Received 2023-05-18; accepted 2023-07-31