Idle Strategy of Smart Monkey to Enhance Testing Operable GUI Regions

Bingyi Cui[1, 2, *], Long Zhang[1, 2, *], Chenglong Sun[3], Zhenyu Zhang[1]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China.
[2] University of Chinese Academy of Sciences, Beijing, China.
[3] Shopee Pte. Ltd., Singapore.

Corresponding author. Email: zhangzy@ios.ac.cn
Manuscript submitted August 20, 2022; accepted November 14, 2022.
doi: 10.17706/jsw.18.3.143-158

Abstract: Graphical User Interface (GUI) testing is an important approach to ensuring software applications' quality. The rendered GUI screen contains operable regions that can be triggered when certain events are applied to these areas. The previous traditional testing methods cannot efficiently identify the GUI area of this operation and generate the sequence of events. Smart Monkey is based on computer vision techniques, which can utilize several basic visual features to confirm the real operable GUI regions. In this paper, we propose an idle strategy of Smart Monkey for enhancing GUI testing. It can use a combination of Monkey and Smart Monkey to achieve high accuracy and efficiency. We implemented the improved technique as an Android testing tool. Then we conduct experiments on 14 real-world applications, comparing with Monkey and Smart Monkey methods, respectively. The results show that it can more efficiently identify operable regions to generate event sequences.

Keywords: Android UI testing, rendered GUI, monkey test

1. Introduction

Mobile Applications are widely used in our digital lives. The criteria for the quality and stability of Android apps are becoming increasingly necessary as the development and iteration of Android applications progress at a rapid pace. The application's Graphical User Interface (GUI) is an important part of ensuring a great user experience, so efficient and high-quality GUI testing is important to ensure the quality of Android apps before being delivered to users.

There are various frameworks for automating GUI testing on the market, including Monkey [1], MonkeyRunner [2], UIAutomator [3], Robotium [4], and Appium [5]. They're great open-source frameworks for ease of use, support for numerous platforms, ensuring a higher quality minimal baseline, lowering expenses, and simplifying the overall testing process.

Monkey test [1] is the most widely used toolkit from Google, due to its excellent compatibility and ease of use on different Android platforms. It sends a completely random sequence of UI events to the test application, with no predetermined rules. However, the current monkey testing technologies have significant drawbacks. There are a lot of modules (or possibly involving complex logic) deep within the structure of the UI interface. The id and class name of some modules may change during application development. Another scenario is when the structural relationship between certain modules changes, such as when a module is transferred to a different parent node. If the above changes occur in the application, the software's interface will remain the

same, but it will have a higher impact on the efficiency and success rate of monkey tests. Due to the completely random exploration of the Monkey test, there is no guarantee that traversal explores all GUI elements.

Previous work [6] has shown that monkey performs poorly in tests on WeChat, a very popular software in China. Most of the events generated by Monkey are redundant, such as repetitive execution of no-code behavior, and its code coverage and operational efficiency are also low, resulting in a large testing overhead.

The user's attention will be drawn to the rendered GUI interface, which will then prompt them as to which regions can be manipulated. Human perception systems and psychological activities are very sensitive, especially to obvious visual cues and application interface modifications. Therefore, our main idea is to try to capture the region that may be changed in the GUI interface by detecting conspicuous areas such as color, intensity, texture, and so on. The method proposed in our previous work has been shown to have better accuracy and efficiency than the traditional Monkey technique. However, Smart Monkey may take longer to detect operable areas in some cases. If some applications need to be tested for a short time or there are few operable areas in the interface, then we may miss some key GUI operation events, resulting in a significant waste of testing overhead. This paper proposes a new strategy to improve the Smart Monkey method‘s inadequacies. We can adopt the Monkey method in the idle time of Smart Monkey operation. Because Smart Monkey needs some time to calculate the salient region, and Monkey can generate a certain number of operation events for testing during this time, minimizing any unnecessary waiting. We conduct experiments on 14 widely used real-world mobile applications. The results demonstrate that Smart Monkey is more effective than Android's Monkey, with 34.96 % of incorrectly detecting the operable region on the screen of these mobile apps, and the first crash failure can be detected earlier. At the same time, Smart Monkey with the idle strategy improves the average detection hit rate of the operation area by 13.21% compared with the method without the idle strategy. The results also show that both the operation hit rate and unique hits per second are greatly improved compared with our previous method.

The main contribution of this paper is threefold:

- It aims to find operable region candidates in rendered GUIs and verify them through specific GUI events.
- The experiment indicates that our method can discover operable region candidates in a rendered GUI and then confirm these possibilities with concrete GUI events, effectively increasing the probability of generating a series of events that can cause the GUIs to respond.
- The results of the experiments reveal that the Smart Monkey technology that uses the idle technique is more efficient than the technology that does not. The operating area's average hit rate has grown by 13.21 %, while the execution duration has remained unchanged.

The rest of this paper is organized as follows. The motivation of this work is presented in Section 2. The motivation of our technique is presented in Section 3. Sections 4 and 5 present the experiment and data analysis of “Smart Monkey”. We presented related works in Section 6 and the conclusion of this paper in Section 7.

2. Motivation

The graphical user interface (GUI) is the most intuitive and important way for a user to interact with software. As a result, in recent years, application developers have paid more attention to the software's GUI.

2.1. GUI Testing

For the traditional Android GUI layout interface, the record-and-replay [4] is the mainstream method for testing these programs. Most applications have many GUI widgets in fixed positions, and this method can be used to control each widget in turn using a test script. However, the graphics engine[7][8][9] renders the full image in gaming apps, which has no set layout structure and is constantly refreshed. Existing solutions

struggle to deal with operational items whose appearance, amount, location, and other features cannot be recognized statically or dynamically during record and replay testing. PUMA [10], for example, is an enhanced version of Monkeys that uses UI-Automator to query the GUI widget layout during runtime. UI-Automator (or DUMP) is unable to query the rendered GUI information.

There are also enhancement versions of Monkeys such as PUMA [10], which uses UI-Automator to query the GUI widget structure at runtime. However, it cannot query the rendered GUI information through UI-Automator (or DUMP for earlier versions of Android). As a result, existing automated testing techniques are unable to handle gaming apps that primarily rely on displayed GUIs.

2.2. Our Insight

The GUI interface of the software should aesthetically try to attract the user and thus gain their attention Human players are also often drawn to attractive elements in still images during gameplay, known as spatial attention [11]. Furthermore, psychological research shows that the human perceptual system is sensitive to visual contrasts such as color, intensity, and texture.

Our previously proposed Smart Monkey can detect operable GUI regions by applying computer vision techniques. Monkey test generates two-dimensional coordinates to construct a user event flow sequence at random in the presented graphical interface. However, there are some clear drawbacks to this strategy, such as the fact that calculating and detecting key regions can take a lengthy time. This method is inefficient in applications that just need to be evaluated for a short period of time or have a few operational regions of the interface. Therefore, we consider if we try to combine this method and Monkey test, whether this combined method can improve the test efficiency, is worth exploring. Next, we will propose an idea to improve the above-mentioned possible deficiencies.

3. Our Technique

In Section III, we present the idle strategy of Smart Monkey and the overall architecture of our method, then explain the detection algorithm.

3.1. Idle Strategy

Smart Monkey is a technology that uses the monkey testing method as its foundation. It can read and analyze the visual content generated on the screen using computer vision technology, and select the picture that is most likely to be a button or label to click or drag. But Smart Monkey takes a long time to calculate key regions, yet many applications require short testing events for some operable regions that emerge seldom. We could miss certain key GUI events if we just halt the testing tool to wait for saliency detection to complete.

As Fig. 1 shown, we propose an idle strategy to overcome the disadvantage. First, we create a new GUI screen, causing our approach to take a picture of the current screen and transmit it to the saliency region detecting server over an *adb* connection and the http protocol [1]. However, Lin et al. [12] used the camera to capture the image on Android phone. The snapshot is received by the host server, which then initiates Saliency Detecting. The Smart Monkey job is executed by one Thread, while Monkey is executed by the other. Two threads start at the same time, and when the Smart Monkey thread finishes calculating, it stops the Monkey thread and both threads store operation events into a ring buffer. We create a consumer thread that picks up operation events from the ring buffer regularly and sends them to the android device over an *adb* connection. If a GUI screen change is triggered by an event, we shall clean the ring buffer and thread status to prepare the server for the next operation.

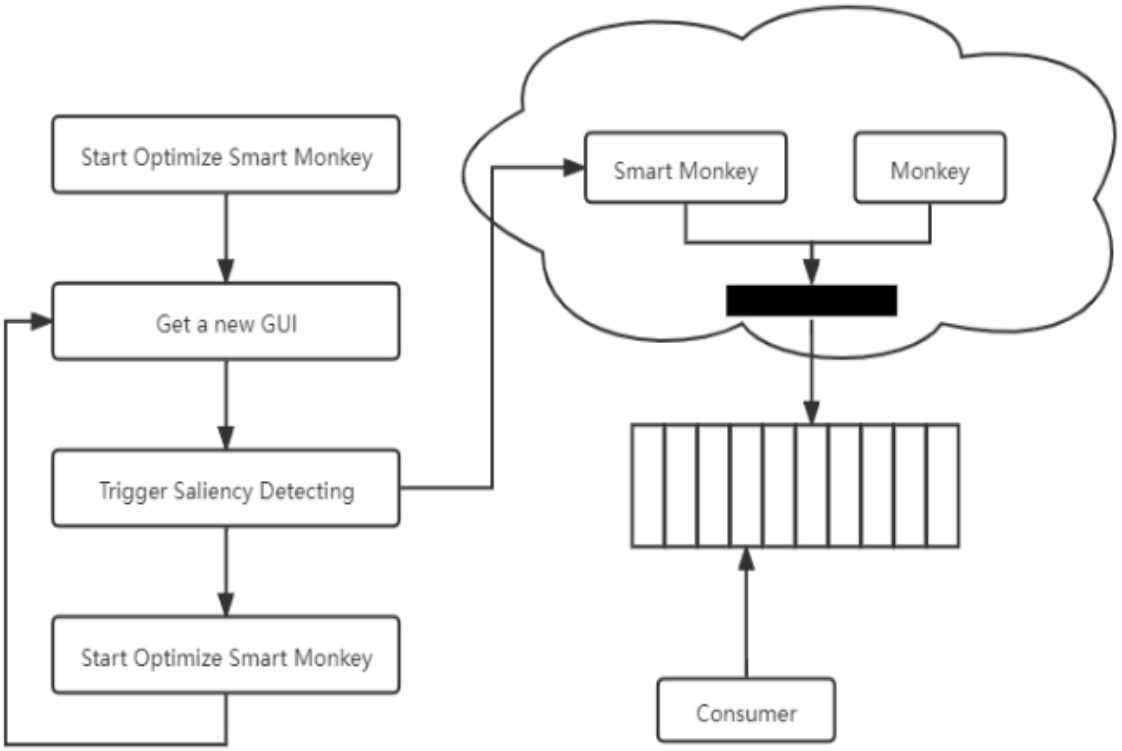


Fig. 1. The structure of Idle Strategy.

3.2. Smart Monkey and Monkey

As part of our method, Smart Monkey takes a screenshot of the app under test and the class name of the current Activity as part of our approach. It's then transferred to the host computer's technical components via the *adb* [13] or *http* protocol [1]. Next, if encountering a new GUI interface currently, this method uses the region detection technique to find the operable region and indexes the GUI state to save the operation result. And if the GUI interface has been identified in the last work, this technology will generate events based on the previous index identification results, which is also the same completely random and iterative exploration strategy.

When exploring the state, Smart Monkey can detect and calculate the operation area of the current GUI state and randomly select an operation area that has not been selected. However, the Monkey tool randomly generates 2D coordinates within the size of the GUI interface. Both methods then send a specific type of event (for example, touch, drag, or click) to random coordinates within that actionable area. The sequence of events they generate is stored in a ring buffer.

3.3. Saliency Detection Algorithms

Our method applies three saliency detection algorithms to detect operable region candidates, by exploiting three fundamental visual features, namely color, density, and texture. Gray spatial attention model [11] (GA algorithm) utilizes density features in grayscale images, and the color spatial attention model [14] (CA algorithm) uses both density and color features. The Spectral Residual Model [15] (SR algorithm) finds that the log spectrum of many images is very similar, and this residual value part is the salient information that can attract people's attention. Therefore, the SR algorithm removes similar spectral information to calculate its residual value. The relationship between the saliency algorithm and the basic visual features used is shown in Table 1.

Table 1. Visual Features and Their Detection Algorithms

Saliency detection algorithms	Visual Features
GA [11]	density
CA [14]	density color
SR [15]	texture

As listed in Fig. 2, the purpose of the saliency detection algorithm is to detect actionable regions from still images [1]. This algorithm of Smart Monkey is implemented using the OpenCV framework and Java and integrated into our tool.

```
function SaliDtc (method)
     @INPUT:  x[] - Image
     @OUTPUT: y[] - Points

01  switch method:
02    case GA: y ← call GA (x)
03    case CA: y ← call CA (x)
04    case SR: y ← call SR (x)
05  return y
```

```
function GenSaliTbl (x)
  @INPUT:  x[] - Image
  @OUTPUT: s[] - Saliency Image

06 for r, c in [1, nRow], [1, nCol]:
07   Hist[x[r][c]]++
08 for outlevel in Hist:
09   for inlevel in Hist:
10     Map[outlevel] ← d[r][c]+
         Hist[level]*call dist(outlevel,inlevel)
11 for r, c in [1, nRow], [1, nCol]:
12   for level in Hist
13     s[r][c] ← Map(x[r][c])
14 return s
```

```
function SaliCan (s)
  @INPUT:  s - Saliency Image
           THRESHOLD
  @OUTPUT: y[] - Points

15  while  y[] is not full
16     r, c ← ran(1, nRow), ran(1, nCol)
17     if s[r][c] < THRESHOLD
18       y ← call cancat(y, d)
19  return y
```

```
function GA
  @INPUT:  x[] - Image
  @OUTPUT: y[] - Points

20  nRow, nCol ← size of x
21  g ← call gray(x)
22  s ← call GenSaliTbl (g)
23  y ← call SaliCan (s)
24  return y
```

```
function CA
  @INPUT:  x[] - Image
  @OUTPUT: y[] - Points

25  l ← call Lab (x)
26  s ← call GenSaliTbl (l)
27  y ← call SaliCan (s)
28  return y
```

```
function SR
  @INPUT:  x[] - Image
  @OUTPUT: y[] - Points

29  g ← call gausian(x)
30  x' ← call fft(x)
31  x'' ← x' - g
32  s ← call ifft (x'')
33  y ← call SaliCan(s)
34  return y
```

Fig. 2. The pseudo-code of Saliency algorithm.

The saliency map of an image is based on measuring the color contrast between image pixels. It is the weighted distance through the pixel value of the image on the histogram of gray or color values of the entire image. For a pixel I_k in an image I, its saliency value can be defined as Eq. (1).

$$S(I_k) = \sum_{\forall I_i \in I} D(I_k, I_i) \tag{1}$$

When we restructured the equation, calculate the frequency of the pixels, and keep it in the equation, the above formula can be converted to the following,

$$\begin{aligned} S(I_k) &= D(I_k, I_1) + D(I_k, I_2) + \cdots + D(I_k, I_N) \\ &= \sum_{j=1}^{n} \left[f_j D(c_l, c_i) \right] \end{aligned} \tag{2}$$

We calculate the frequency of the pixel value and save it in the hash-map, which can be directly calculated in the subsequent calculation of the saliency value to reduce the complexity. In addition, we use a random

strategy to identify the most likely operable points in the saliency image.

The GA algorithm calculates the grayscale distance between each pixel as a map for each saliency value. The CA algorithm calculates the hash table of the saliency values between each pixel through the LC algorithm for the color values in $L \times a \times b$ color space. The SR algorithm is calculated in the frequency domain. It first calculates the Fourier spectrum of the image, and then subtracts Gaussian noise from the source spectral image to obtain the saliency spectrum image. And we apply inverse fast Fourier transformation [15] to this image.

4. Experiment

In this study, we sought to understand how the adoption of Monkey during idle periods affects the effectiveness of testing. We address five main research questions in our paper through experiments.

RQ1: Can Smart Monkey effectively detect actionable widgets in mobile applications?

RQ2: Is Smart Monkey more effective than the built-in monkey of Android?

RQ3: Is it useful that run Monkey during Smart Monkey's idle time?

RQ4: Under equal operation events, does the combination of Smart Monkey and Monkey is more efficient than Smart Monkey or Monkey test?

RQ5: Which feature is the key part to increase testing effectiveness?

4.1. Evaluative Dimensions

In order to answer the questions proposed above, we take several evaluative dimensions, namely, Hit, Unique Hit, Hit Ratio, Crash Time, Screen Cover Ratio.

- Hit is the number of generated operation events that trigger operable objects.
- Unique Hit is the number of operable objects triggered by generated operation events.
- Hit Ratio is the ratio between Unique Hit and all operable objects.
- Crash Time is the period of time from program start to crash end.
- Screen Cover Ratio is the one between the triggered screen and all screens.

The number of hits indicates how effective a tool is in generating valid operation events. We argue the tool is inefficient in evaluating a program if most events trigger the same operable item. As a result, we'll require a new dimension called Hit Ratio, which refers to the fraction of covered operable items on a single screen. Because a tool seldom causes a program to crash, Crash Time may not be the best metric to use when measuring a tool's effectiveness. We present the Screen Cover Ratio as a metric for determining the effectiveness of testing.

4.2. Experimental Setup

We evaluate the effectiveness of our method on 14 representative real-world applications, compared to Monkey that built in Android. We used a mobile phone running this Android OS 5.1 and a host with JDK, OpenCV and Android SDK. Four parts of our experiment are as follows.

In the first experiment, we compared the Smart Monkey with Android's built-in monkey technology to test the effectiveness of its method. We ran 14 applications five times on three devices for each technique to calculate the average number of active actions (e.g., clicks, taps) throughout a three-minute period. Furthermore, we also compared the relative effectiveness of three algorithms. Similarly, we ran the same application 5 times on the same device to calculate the mean of effective operations.

In the second experiment, we tested several game applications separately and used the tool to record the first crash's time. so we could compare the performance of our tool and the built-in Monkey tool.

In our third experiment, we want to figure out how adopting Monkey during Smart Monkey's affect the effectiveness of testing. Monkey's strategy is to generate a certain number of operation events and send them

to the mobile device at once, while Smart Monkey takes up much more time to calculate the saliency region. To get the convictive conclusion, we should measure all the technologies under the same condition that they should generate an approximate number of operation events. Because if the ratio of generated GUI events between Monkey and Smart Monkey were too large, the Smart Monkey with idle strategy will degenerate to Monkey. So, we run all of programs by using single Monkey, by using Smart Monkey with mixture algorithm, and by Smart Monkey with idle strategy. In the end, we record the Hit and Crash Time.

Our previous work shows that Smart Monkey's hit ratio no longer apparently increases if the number of operation events is more than10. So, we adopt the same strategy that limits the number of operation events generated by Monkey and by Smart Monkey to 10 in one testing circle.

In our fourth experiment, we want to make sure of the affluence of saliency detecting algorithm. Therefore, we run 14 programs by using the combination of Monkey and Smart Monkey, and Smart Monkey adopts single saliency region detecting algorithm, namely LC or SR. So, we could know which property is the key part to affect the testing effectiveness.

We developed our method as an android tool, which is built by Maven [16].

5. Analysis

In this section, the results of experiments are presented to answer the research questions.

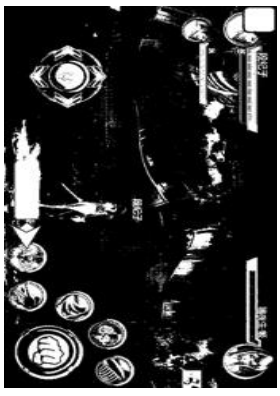
5.1. Answering RQ1

Fig. 3 contains examples of four applications, where the plots (a), (c), (e), (g) are source images, and the plots (b), (d), (f), (h) are saliency map binary images. The green dots[1] in each graph represent the saliency detection results.

And the green circles are the saliency point, the b, d, f, h pictures all are the binary saliency map.

The first four pictures are from the interface of the Android game Final Fight, including two sets of test images. The screen is horizontal. The plots (b) and (d) are the saliency images of the plots (a) and (c), respectively. After the calculation, a salient area will be marked in the image, and green dots will be randomly placed in the area. Plot (a) contains 6 significant regions and plot (c) contains 10. The last four pictures are the interface from the Android game Piano Tiles 2, and the screen is vertical. We notice that the principal color in the interface changes from (e) to (g), and the saliency image reverses accordingly. However, the actionable areas of the two images are similar, such as the Home and Hall buttons below the recognition screen. This shows that this detection algorithm is robust. All computation costs take no more than 800ms and the detection algorithm has a certain error rate, which is acceptable for the performance of the interactive system.

(a) Screenshot of Final Fight fighting scene

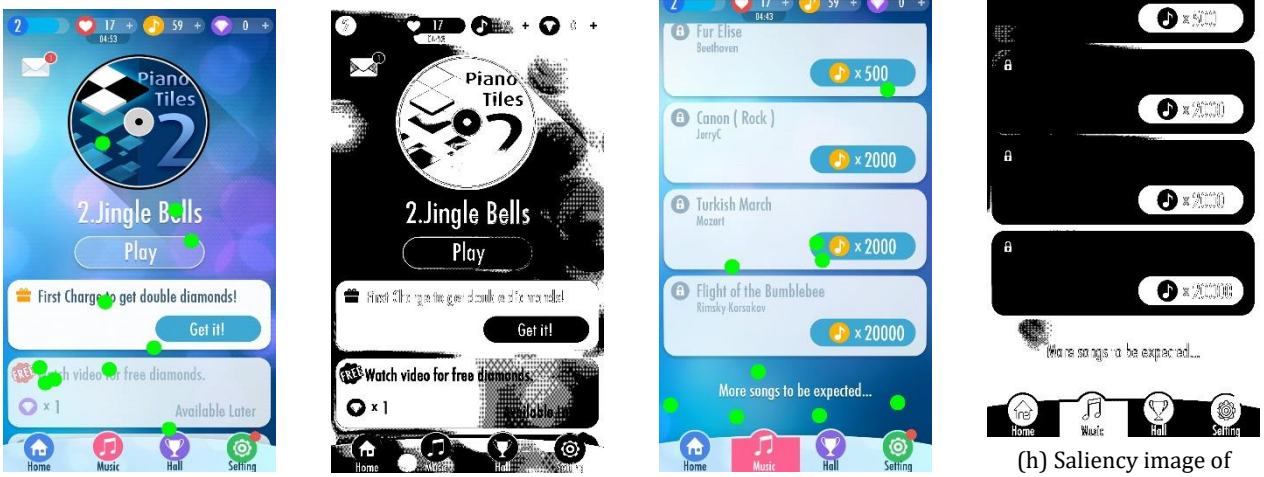


(b) Saliency image of Final Fight fighting scene

(c) Screenshot of Final Fight choose hero scene

(d) Saliency image of Final Fight choose hero scene

[1] Green dots may be hard to see on grayscale images.



(e) Screenshot of Piano Tiles start a scene

(f) Saliency image of Piano Tiles start a scene

(g) Screenshot of Piano Tiles music choose scene

(h) Saliency image of Piano Tiles music choose scene

Fig. 3. Some saliency detection in several android applications. The a, c, e, g pictures all are the source image.

Our experiments then measure the percentage of effective hits for each application. The results are shown in Table 2. We experimentally obtain the number of images for each application, and an overview of its operable objects. In addition, the number of hits and unique hits in actionable positions is the result of running our method. The number of unique hits refers to the number of operations that are not repeated for the operable area, and the multiple tests of the same operable region are counted only once. Hit Ratio is calculated by dividing the total number of actionable objects by the number of unique hits.

Table 2. The Coverage Ratio on Applications (Smart Monkey)

	Image Num	Total Operable Objects	Hit Num	Hit Num Uniq	Hit Ratio
WeChat	4	49	27.33	17.67	36.06%
Twitter	3	39	23.67	10.67	27.35%
Temple Run	3	22	17.33	4.67	21.21%
Snake	18	114	89	46	40.35%
Sanbei	9	102	53	36	35.29%
Momo	9	115	88	53	46.08%
Mobike	15	127	79	45	35.43%
Jingdong	5	103	35.33	26.33	25.57%
Instagram	3	27	21.00	9.67	35.80%
GuitarTune	5	55	28.33	18.33	33.33%
GameDev	4	32	8.00	7.67	23.96%
Flickr	4	25	23.00	9.67	38.67%
Deadlyracing	5	23	14.67	12.67	55.07%
ClashOfClan	23	368	179	130	35.33%

For example, the first row is the result of Smart Monkey running 3 images on Wechat. There are 49 operable objects in total and 44operations. On average 23.67 of them hit operable objects, and 10.67 operations are unique. So, the hit ratio is 27.35%.

Our tool attains the best results on *Deadlyracing* with a hit ratio of 55.07%, while the worst results on *Temple* with a hit ratio 21.21%. The average hit ratio in these fourteen applications is 34.96%. Overall, the above results can show that our proposed method is effective.

Thus, we can answer *RQ1* that our technique is effective to identifying operable widgets from the rendered images of mobile game screenshots.

5.2. Answering RQ2

To analyze the relative effectiveness of our tool, we conducted an experiment comparing our method with the Android built-in Monkey tool, and counted the number of valid operations within three minutes. The experimental results are shown in Table 3. It includes the effective operands of our tool and Monkey, and the incremental ratio of Smart Monkey in comparison to Monkey.

For example, in the first row of the table, the test of Smart Monkey on Wechat shows that it can perform 11.2 effective operations in 3 minutes, and the test result of Monkey test is 5.6. Therefore, the performance of this method on these applications is improved by 100% compared to the Monkey. The result of other apps can be explained similarly. Overall, the result of the incremental ratio is positive, with an average of 79.60%.

Table 3. The Valid Operations in 3 Minutes

	Smart Monkey	Monkey	Incremental Ratio
WeChat	11.2	5.6	100.00%
Twitter	12.4	8.2	51.22%
Temple Run	11.6	11.2	3.57%
Snake	11.3	6.1	85.25%
Sanbei	13.7	10.0	37%
Momo	18.8	7.9	137.97%
Mobike	10.4	3.6	188.89%
Jingdong	15.8	6.4	146.88%
Instagram	14.0	9.6	45.83%
GuitarTune	14.6	8.2	78.04%
GameDev	10.2	4.8	112.50%
Flickr	16.0	14.0	14.28%
Deadlyracing	12.6	10.8	16.67%
ClashOfClan	21.0	10.7	96.26%

To test the failure detection ability of our method, we performed an experiment to investigate the time it takes to analyze the first failure, running our tool and Monkey on each application. We calculate the mean time to failure in seconds for each of the five runs. The performance of the two methods is shown in Fig. 4. Our tool can find crashing errors faster in all four applications, compared to monkeys. And three of the programs tested have an improvement rate of 8%.

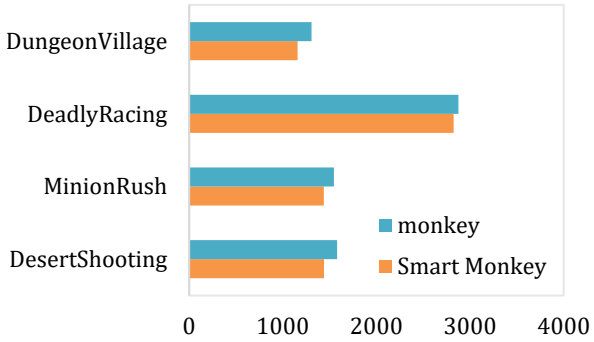


Fig. 4. The crash time (unit: second) of the application running on Smart Monkey and monkey separately.

Therefore, the above experimental results show that our technique is more effective than Monkey in exposing crashing application failures when testing mobile game applications with rendered GUIs.

5.3. Answering RQ3

Table 4. The Coverage Ratio on Applications (Smart Monkey with Idle)

	Image Num	Total Operable Objects	Hit Num	Hit Num Uniq	Hit Ratio (Smart Monkey)	Hit Ratio (with idle)
Wechat	4	49	32	20.3	36.06%	41.43%
Twitter	3	39	40.3	17.6	27.35%	45.13%
Temple Run	3	22	18.3	15.75	21.21%	71.59%
Snake	18	114	106	73	40.35%	64.04%
Sanbei	9	102	74	42	35.29%	41.18%
Momo	9	115	97	66	46.08%	57.39%
Mobike	15	127	83	59	35.43%	46.47%
Jingdong	5	103	39	27.33	25.57%	26.53%
Instagram	3	27	31.00	12.7	35.80%	47.04%
GuitarTune	5	55	33.6	23	33.33%	41.82%
GameDev	4	32	8.00	13.35	23.96%	41.72%
Flickr	4	25	24.00	11.7	38.67%	46.8%
Deadlyracing	5	23	17	14.2	55.07%	61.74%
ClashOfClan	23	368	255	153	35.33%	41.58%

Table 4 is the statistical information of fourteen different applications after running by Smart Monkey with the idle Monkey strategy. The first five columns of Table IV are similar to that of Table II. The sixth and seventh columns in Table 4 they are Hit Ratio of the previous method and the improved method using the idle strategy, respectively. For instance, the first row is the Smart Monkey with idle strategy running on Momo's 9 screenshots. There are a total 115 operable objects. And our tool has succeeded to predict 88 valid operations and 66 operations are unique. so, the hit ratio is 57.39% ($=\frac{Hit\ Uniq}{Total\ Operable\ Objects}$).

Among the 14 applications in the experiment, the improved method using the idle strategy has achieved a high hit ratio, which is a significant improvement compared to the previous method. We can calculate the average hit ratio of only adopting Smart Monkey is 34.96% from the sixth column. The seventh column of Table V is our tool with idle strategy, the average hit ratio of the optimized tool by adding idle strategy is 48.17%. We can conclude that it could improve the effectiveness of Smart Monkey by adopting Monkey during Smart Monkey idle time, and cause no time increase in execution.

Suppose if the new strategy didn't increase the Unique Hit, the result will be equal to Smart Monkey, and in other cases, adopting this strategy could improve Smart Monkey under the measure of Hit Ratio. The reason why the Hit ratio doesn't significantly improve lies in that Smart Monkey covers most of the operation events generated by Monkey during idle.

5.4. Answering RQ4

Table 4. Hit Ratio Incremental

Application Name	Monkey Hit	Hit of Smart Monkey with idle	Monkey Hit Ratio	Hit Ratio of Smart Monkey with idle	Incremental Ratio
Twitter	14	40.3	14.34%	44.33%	209.14%
Instagram	14	31	8.81%	24.87%	182.28%
Momo	21	88	23.48%	46.08%	96.25%
Snake	43	89	23.68%	40.35%	70.40%
ClashOfClan	77	179	17.89%	35.33%	97.48%
Sanbei	42	53	17.65%	35.29%	99.94%
Mobike	39	79	22.05%	35.43%	60.68%
Flickr	13.39	19.6	18.45%	35.75%	93.77%

Wechat	13	32	16.87%	37.80%	124.12%
GuitarTune	6.3	15.7	15.54%	35.97%	131.47%
Deadlyracing	4	11	20.37%	73.33%	259.99%
Temple Run	5.3	16.7	15.3%	61.14%	266.11%
GameDev	7.14	18.4	14.93%	22.37%	49.83%
Jingdong	19	39	10.27%	25.41%	147.41%

We compare our new tool with the Monkey tool to find out the effectiveness of our new tool. In Table V, the second and third columns are Monkeys hits and Smart Monkey hits. The fourth and fifth columns separately are Monkey's hit ratio and that of Smart Monkey with idle. The last column is an incremental ratio, which can be calculated by formula.

$$\frac{Smartmonkey\ idle\ hit\ ratio - Monkey\ hit\ ratio}{Monkey\ hit\ ratio}.$$

We can figure out that Smart Monkey with idle strategy can generate more valid operations. In our previous work, the average incremental ratio is 63.22%, and our new tools incremental ratio is 134.92%. We can draw the conclusion that our new tool is much better than Monkey and better than our previous work.

Table 5. Screen Coverage Ratio

Application Name	Smart Monkey	Smart Monkey with idle	increase
Twitter	6	8	16.67%
Instagram	5	9	80%
Momo	7	12	71.43%
Snake	4	6	50.00%
ClashOfClan	6	7.2	20.00%
Sanbei	4	5	25.00%
Mobike	10	11	10.00%
Flickr	13	15	15.38%
Wechat	6	11	83.33%
GuitarTune	7	8	14.29%
Deadlyracing	6	7	16.67%
Temple Run	5	7	40%
GameDev	5	8	60%
Jingdong	7	10	42.86%

While in most cases, popular applications are fully tested before handover, there is of small possibility to trigger a program crash under limited testing times. Therefore, we need other measurements to evaluate tool's testing efficiency. We collect all the screens of each application triggered by our tool and Smart Monkey. Table 5 is the triggered screen information; the second column is the number of triggered screens by Smart Monkey and the third column is the one by Smart Monkey with idle. From the fourth column, we can find out that Smart Monkey with idle can cover more screens than Smart Monkey. In a review of the testing result, we find that some screen contains too many saliency regions, but they are not operable. what's worse only a few regions are operable. In such cases, only adopting Smart Monkey may stay on a screen too long, while Monkey generates points faster and has a certain possibility to trigger screen change, so the result can improve.

We wonder how idle strategy can affect Smart Monkey. Then we collect the valid unique hits per second of Smart Monkey and Smart Monkey with idle. In table VII, the first column is the unique hits per second of Smart Monkey, the second column is the one of Smart Monkey with idle. The third column is the incremental unique hit per second=($\frac{Smart\ monkey\ with\ idle - Smart\ monkey}{Smart\ monkey}$). We can find that adopting idle strategy can significantly improve Smart Monkey in unique hits per second, which means Smart Monkey with idle is more effective. The reason why adopting idle can improve Smart Monkey lies in that we fully take advantage of the idle time before Smart Monkey returns possible operable operations.

5.5. Answering RQ5

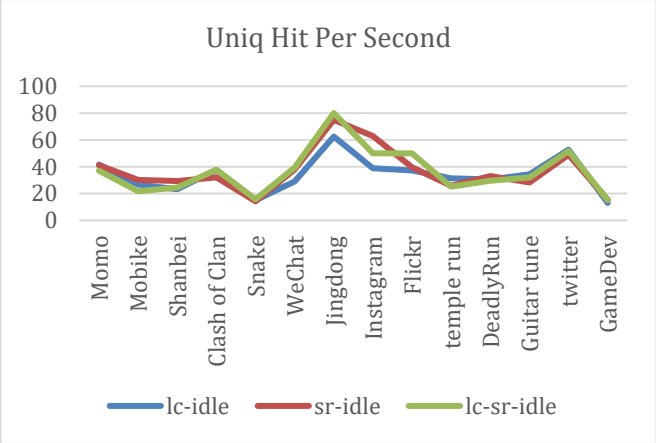


Fig. 5. Saliency Algorithm Impact.

Smart Monkey includes two different algorithms to predict saliency regions, namely, LC and SR. In normal processing, we consider both of them and adopt the predicting result according to their processing time. To figure out whether different algorithms have different effectiveness in detecting saliency regions, we carry out a comparison experiment, and the result is in Fig. 5. The vertical axis is the value of a unique hit per second, the horizontal axis is five applications. We can find out the shapes of the three strategies are similar, and the algorithm does insignificant influence in unique hit per second, and it indicates that the algorithm is not the key feature to affect Smart Monkeys effectiveness.

5.6. Case Study

Fig. 6. Case study.

For the convenience of analysis, we use java library log4j to remark the points generated by Monkey and Smart Monkey. The blue points are generated by Monkey and the green points are generated by Smart Monkey. The numbers in points are the order in which they are generated. For instance, the green point with 4 located in the top middle screen is the 4th point generated by Smart Monkey. The explanation can also be applied to other points.

If we only use Monkey, the operable points are point 2, point 3, and point 5. The valid points generated by Smart Monkey are point 1, point 3, point 5, and point 8. Unfortunately, blue point 5 clicked the left-below button, and this would trigger screen change. What's more, the new tool will abandon all results generated by Smart Monkey.

Suppose the execution time of one point is 10 ms, it will cost 100 ms for Smart Monkey to generate 10 points. Monkey will generate 10 points and there is no screen change happened. Therefore, we can calculate

that the unique hits per second of Monkey is $30 = 3/0.1$, Smart Monkey is $20 = 4/0.2$ and Smart Monkey with idle is $35 = 7/0.2$.

In this case, blue point 5 triggered screen change, so the costed time is $50ms = 10ms * 5$ and all three operable points generated during idle time are executed. The unique hit per second is $60 = 3/0.5$, which is bigger than that of Smart Monkey with idle.

Table 6. Idle Incremental in Unit Time

Application Name	Smart Monkey	Smart Monkey with idle	Incremental Unique Hit per Second
Twitter	48.4	51.11	5.6%
Instagram	11.70	19.17	63.85%
Momo	32.7	37.08	13.39%
Snake	13.24	21.68	63.75%
ClashOfClan	15.05	24.48	62.66%
Sanbei	22.43	37.76	68.35%
Mobike	14.49	15.75	8.70%
Flickr	15.38	23.03	49.74%
Wechat	14.03	25.45	81.40%
GuitarTune	23.38	31.5	34.73%
Deadlyracing	27.04	31.06	14.87%
Temple Run	19.75	27.45	38.99%
GameDev	11.76	15.38	30.78%
Jingdong	14.56	20.58	41.35%

6. Related Work

6.1. Android UI Testing

Graphical user interface (GUI) testing [1] has always been an important method to ensure the quality of Android applications. It can simulate the interaction behavior of real users and explore potential program defects. Monkey [1] is a purely randomized Android test generation tool presented by Google that generates random streams of UI events with no model construction. As a part of the Android developers' toolkit, users have not required any additional installation effort. Machiry et.al present Dynodroid [17], also based on random exploration. It can choose both the least frequently chosen events (Frequency strategy) and context (BiasedRandom strategy). Therefore, more contextually relevant events will be selected more frequently. WeChat team develops a new approach named WCTester [6], combining several strategies to inherit the advantages of Monkey while improving its main constraints.

Model-based testing is a widely used testing approach. Mao et.al present an evolutionary-testing-based test generation tool for Android UI testing, called Sapienz [19]. It uses a genetic algorithm to evolve generated seed input sequences to select the optimal test suites with short input sequences that maximize code coverage and fault detection. Su *et. al.* presented Stoat [21], a model-based GUI testing tool for Android apps. Stoat used Gibbs sampling to search for the optimized model and guide test generation from mutated models. Li et.al introduced DroidBot [22], a lightweight and model-based Android UI testing tool, which generates UI-guided test inputs based on a state-transition model.

PUMA [10] is a framework presented by Hao *et al*., which can implement any dynamic analysis on Android apps based on its same basic random exploration as Monkey. Amalfitano et.al introduced GUIRipper [23] which based on a user-interface driven ripper. It can automatically explore GUI and dynamically build a model of the app under test by crawling it from a starting state. Yang et.al introduced ORBIT [24], which can statically analyze the app's source code and find the relevant UI events for a specific activity. It could generate more relevant inputs, so is more efficient than GUIRipper.

Systematic exploration strategy uses more sophisticated techniques such as symbolic execution and

evolutionary algorithms. It has more exploration capabilities and could provide specific input to reveal certain applications behavior. N. Mirzaei *et al.* [25] used symbolic execution to improve the performance of android testing. T. Azim *et al.* [26] proposed a strategy of Depth-first Exploration, which can explore the components of activities in a more systematic way, to achieve the effect of imitating user behavior. Mahmood et.al presents EvoDroid[27], an evolutionary algorithm to generate relevant inputs for system testing of Android apps. It uses the test input sequence to represent individuals and implements a fitness function to maximize coverage. Anand et al. introduced ACTEve [28], a concolic-testing tool that could handle both system and UI events. It symbolically tracks events from the point where they generate to the point where they are ultimately handled.

Computer vision is an interdisciplinary scientific field that deals with how computers can gain high-level understanding from digital images or videos [29]. Chang *et al.* [30] used computer vision techniques to improve GUI testing results. They implemented a testing tool called Sikuli, which takes screenshots of the target and identifies graphics modules in desktop applications.

6.2. Saliency Detection

Visual saliency detection refers to simulating human visual characteristics through intelligent algorithms to extract salient areas in images. It has been widely used in image segmentation, video compression, object detection, and other fields.

There are a lot of other saliency detection algorithms, divided into two kinds, pixel space or spectral-based [11], and feature-based [32]. Ltti *et al.* [34] proposed a visual attention model based on Gaussian pyramid fusion of image color, brightness and orientation features. Xie *et al.* [33] exploited low- and mid-level cues that based on Bayesian framework to detect saliency regions. Hou *et al.* [15] studied the spectral residual contained in the log spectrum of the image frequency domain, which is the salient information in the image that can attract the attention of the human visual system. Cheng *et al*. [35] proposed HC (Histogram-based Contrast), which is a color contrast algorithm based on the color global histogram. The greater the difference in color features between a pixel and other pixels, the higher the significance. The GR (Graph-Regularized) algorithm is a saliency extraction algorithm based on superpixel segmentation and contrast between regions proposed by Yang *et al.* [36].

7. Conclusion

Existing automated mobile testing methods cannot adequately test mobile gaming apps with displayed GUI widgets. Our previous work proposes the Smart Monkey technique, which can use a saliency detection algorithm to identify actionable regions displayed in applications as actionable region candidates. In this study, we propose an idle strategy to improve the existing flaws of Smart Monkey by combining it with Monkey testing techniques. It can avoid long waits for inspections that could potentially miss critical GUI time. We have developed Android testing tools to implement our improved technique and tested it in real applications. The experimental results show that the new strategy can improve the operation hit rate and unique hits per second, confirming that our method is effective in detecting Actionable area aspects of realistic rendering in applications that are accurate and more efficient.

In future work, we can consider combining multiple detection algorithms to improve detection efficiency, and then run our technique on other platforms, such as mobile web applications or cloud test environments.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Bingyi Cui and Long Zhang wrote the paper; Chenglong Sun conducted the research; all authors had approved the final version.

References

[1] Android Developer Website. (2021). Monkey. Retrieved from: https://developer.android.com/studio/test/monkey

[2] Android Developer Website. (2021). Monkeyrunner. Retrieved from: https://developer.android.com/studio/test/monkeyrunner

[3] Android Developer Website. (2021). UI Automator. Retrieved from: https://developer.android.com/training/testing/other-components/ui-automator

[4] Github, Inc. robotium. Retrieved from: http://code.google. com/p/robotium

[5] Sauce Labs. Appiunm. Retrieved from: https://saucelabs. com/appium

[6] Xia, Z. *et al*. (2016). Automated test input generation for Android: Are we really there yet in an industrial case? *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 987–992).

[7] Khronos Group. Opengl. Retrieved from: https://www. opengl.org

[8] Cocos Play. Cocosplay. Retrieved from: http://play.cocos. com

[9] Unity Technologies. Retrieved from: http:// unity3d.com.

[10] Hao, S., Liu, B., Nath, S., Halfond, W., & Govindan, R. (2014). PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (pp. 204–217).

[11] Zhai, Y., & Shah. M. (2006). Visual attention detection in video sequences using spatiotemporal cues. *Proceedings of the 14th Annual ACM International Conference on Multimedia* (pages 815–824).

[12] Lin, Y. D., Chu, E. T., Yu, S. C., & Lai, Y. C. (2014). Improving the accuracy of automated GUI testing for embedded systems. *IEEE Software*, *31(1)*, 39–45.

[13] Android Developer Website. Retrieved from: https://developer.android.com/studio/command-line/adb

[14] Cheng, M., *et al.* (2015). Global contrast based salient region detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *37(3)*, 569–582.

[15] Hou, X., & Zhang, L. (2007). Saliency detection: A spectral residual approach. *Proceedings of 2007 IEEE Conference on Computer Vision and Pattern Recognition*.

[16] The Apache Software Foundation. Retrieved from: http://maven.apache.org

[17] Machiry, A., Tahiliani, R., & Naik, M. (2013). Dynodroid: An input generation system for Android apps. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.*

[18] Zheng, H. B., *et al*. (2017). Automated test input generation for android: Towards getting there in an industrial case. *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track*.

[19] Mao, K., Harman, M., & Jia, Y. (2016). Sapienz: Multi-objective automated testing for Android applications. *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

[20] Mao, K., Harman, M., & Jia, Y. (2017). Crowd intelligence enhances automated mobile testing. *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*.

[21] Su, T., *et al*. (2017). Guided, stochastic model-based GUI testing of Android apps. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*.

[22] Li, Y. C., *et al*. (2017). DroidBot: A lightweight UI-guided test input generator for android. *Proceedings of the* 2017 *IEEE/ACM 39th International Conference on Software Engineering Companion*.

[23] Amalfitano, D., *et al*. (2012). Using GUI ripping for automated testing of android applications. *Proceedings*

of the 27th IEEE/ACM International Conference on Automated Software Engineering.

[24] Yang, W., Prasad, M. R., & Xie, T. (2013). A grey-box approach for automated GUI-model generation of mobile applications. *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering.*

[25] Mirzaei, N., Malek, S., Păsăreanu, C. S., Esfahani, N., & Mahmood, R. (2012). Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, *37(6)*, 1–5.

[26] Azim, T., & Neamtiu, I. (2013). Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, *48(10)*, 641–660.

[27] Mahmood, R., Mirzaei, N., & Malek, S. (2014). EvoDroid: Segmented evolutionary testing of android apps. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[28] Anand, S., Naik, M., Harrold, M. J., & Yang, H. (2012). Automated concolic testing of smartphone apps. Proceedings *of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.*

[29] Wikimedia Foundation. Retrieved from: https://en.wikipedia.org/wiki/Computer_ Vision

[30] Chang, T. H., Yeh, T. R., & Miller, C. (2010). GUI testing using computer vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.

[31] Chang, T. H., Yeh, T. R., & Miller, C. (2001). Sikuli: Using GUI screenshots for search and automation. *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*.

[32] Memon, A. M., Pollack, M. E., & Soffa, M. L. (2001). Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, *27(2)*, 144–155.

[33] Xie, Y., Lu, H., & Yang, M. H. (2013). Bayesian saliency via low and mid-level cues. *Image Processing, IEEE* Transactions, *22(5)*, 1689–1698.

[34] Itti, L., Koch, C., & Niebur, E. (1998). A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *20(11)*, 1254–1259.

[35] Cheng, M. M., *et al.* (2015). Global contrast based salient region detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *37(3)*, 569–582.

[36] Yang, C., Zhang, L., & Lu, H. (2013). Graph-regularized saliency detection with convex-hull-based center prior. *IEEE Signal Processing Letters*, *20(7)*, 637–5640.