GLIB: Towards Automated Test Oracle for Graphically-Rich Applications

Ke Chen*
Fuxi AI Lab in Netease
Hangzhou, China
chenke3@corp.netease.com

Yufei Li*
University of Texas at Dallas
Dallas, USA
yxl190090@utdallas.edu

Yingfeng Chen
Fuxi AI Lab in Netease
Hangzhou, China
chenyingfeng1@corp.netease.com

Changjie Fan
Fuxi AI Lab in Netease
Hangzhou, China
fanchangjie@corp.netease.com

Zhipeng Hu
Fuxi AI Lab in Netease
Hangzhou, China
zphu@corp.netease.com

Wei Yang
University of Texas at Dallas
Dallas, USA
wei.yang@utdallas.edu

ABSTRACT

Graphically-rich applications such as games are ubiquitous with attractive visual effects of Graphical User Interface (GUI) that offers a bridge between software applications and end-users. However, various types of graphical glitches may arise from such GUI complexity and have become one of the main component of software compatibility issues. Our study on bug reports from game development teams in NetEase Inc. indicates that graphical glitches frequently occur during the GUI rendering and severely degrade the quality of graphically-rich applications such as video games. Existing automated testing techniques for such applications focus mainly on generating various GUI test sequences and check whether the test sequences can cause crashes. These techniques require constant human attention to captures non-crashing bugs such as bugs causing graphical glitches. In this paper, we present the first step in automating the test oracle for detecting non-crashing bugs in graphically-rich applications. Specifically, we propose GLIB based on a code-based data augmentation technique to detect game GUI glitches. We perform an evaluation of GLIB on 20 real-world game apps (with bug reports available) and the result shows that GLIB can achieve 100% precision and 99.5% recall in detecting non-crashing bugs such as game GUI glitches. Practical application of GLIB on another 14 real-world games (without bug reports) further demonstrates that GLIB can effectively uncover GUI glitches, with 48 of 53 bugs reported by GLIB having been confirmed and fixed so far.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Neural networks**.

*The first two authors contributed equally to this research.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468586

KEYWORDS

Automated Test Oracle, Game Testing, GUI Testing, Deep Learning

ACM Reference Format:
Ke Chen, Yufei Li, Yingfeng Chen, Changjie Fan, Zhipeng Hu, and Wei Yang. 2021. GLIB: Towards Automated Test Oracle for Graphically-Rich Applications. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468586

1 INTRODUCTION

Graphically-rich applications (also short for apps) have been popular on mobile and personal computer (PC) platforms. With a growing number of complex visual effects such as advanced rendering, light and shadows, animation, and intensive media embedding being used to enhance the quality of GUI (also short for UI) [29], various graphical glitches may occur in the apps and severely impact user experience. Existing automatic UI testing techniques [45] detect bugs by generating test sequences and check whether some crashes are caused. Therefore, these techniques require constant human attention to capture the UI glitch-inducing bugs. However, there are quantities of UI glitches that can severely degrade graphically-rich apps' usability but not induce crashes in practical scenarios. Hence, in this paper, we make a first step in addressing the lack of oracle problem for graphically-rich apps. Specifically, we propose an automated test oracle for detecting UI glitches in game apps.

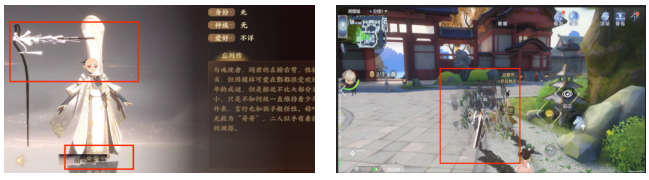


Figure 1: Examples of game UI glitches.

Recent image-based UI testing techniques [29, 44] demonstrate that adding images with versatile UI display issues to the training datasets can help improve the performance of Convolutional Neural Network (CNN)-based detection models in non-game mobile

apps. For example, Owl Eyes [29] designs a heuristic-based data augmentation approach for generating abnormal screenshots on Rico dataset [21] by mimicking the symptom of real-world UI display issues. Its main methodology is to classify UI display issues into five classes and design each issue generation rule according to its features. With a large amount of generated UI screenshots, Owl Eyes improves the effectiveness of detecting UI glitches in non-game apps significantly.

However, we observe that the existing heuristic-based data augmentation approach applied in Owl Eyes cannot accurately reflect the UI glitch issues in graphically-rich applications, especially in game scenarios due to three main reasons. First, their manually-defined rules require human inspection on screenshots of UI glitches and humans may miss certain unnoticeable but important patterns. Moreover, their generation process is to mimic the screenshots of UI glitches, thus the generated images may be infeasible to be generated by the real bugs in the program code. This approximation may cause false positives in the detection process. Last, Owl Eyes mainly focuses on text-related UI display issues, whereas in game scenario, the UI display issues are typically text-irrelevant graphical glitches which may not be generalized by the heuristic rules defined by Owl Eyes.

To address these issues, we propose GLIB, an automated test oracle to detect UI glitch-related bugs. To enable better performance of GLIB, we develop a code-based data augmentation approach to augment the training data for GLIB by injecting the buggy code snippets to the game apps and record the manifestation of the bugs (i.e., UI glitches). In this way, our generated screenshots contain real UI glitches so that the DL model can be trained with more precise datasets and potentially learn subtle patterns that humans may not observe. Moreover, our study of bugs' root causes can guide developers to debug with some empirical knowledge after detecting the UI glitches. Because some UI glitch issues occur in only parts of the UI screen area and human inspectors may miss the issues, we develop a technique based on the saliency map [37] to localize the glitch regions with different bug categories so that the developers can easily determine whether and where our detected images have UI glitches.

To better evaluate the effectiveness of GLIB, we create a testing dataset consisting app screenshots with and without UI glitches from real-world game bug reports. Evaluation on the testing dataset demonstrates that GLIB can achieve 0.9% and 76.7% boost in precision and recall compared to the prediction results of the model trained without data augmentation, leading to 100% precision, 99.5% recall, and 99.8% F-1. Moreover, we evaluate the practical usefulness of GLIB by detecting UI glitches in 14 real-world games with different platforms and engines, the practical application result shows that our model can successfully spot previously undetected UI glitch issues and help developers to fix the bug.

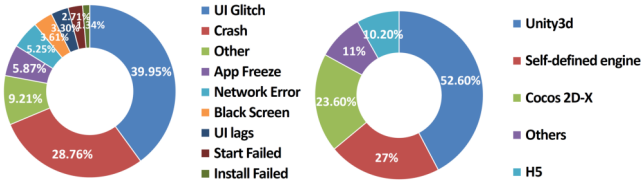
The contributions of this article are as follows:

- Our work[1] [27] is the first to systematically investigate UI glitch issues in real-world graphically-rich apps. We create a large-scale dataset of screenshots with UI glitches and release the data [28] for follow-up studies.
- Based on our characteristic study on the root causes of graphical UI glitches, we propose a code-based training data augmentation approach that can be applied in real-world game apps to generate UI glitches. Our study can also guide developers to find and fix the bug after detecting game UI glitches.
- We propose a CNN-based model for detecting images with UI display issues, and leverage saliency map to localize the glitch region in the UI.

[1]Code to reproduce our experiments is available at https://github.com/GLIB-game/GLIB.git

2 BACKGROUND

Game testing company TestBird [8] collected and tested 11,476 mobile game apps in 2020 and reported 552,851 relevant compatibility issues. According to the statistics of 300 terminals tested for each game, the average number of game compatibility issues is 44 and the average pass rate is 86.41%. TestBird analyzed all these compatibility issues and classified them into 10 categories, namely, *UI glitch, install failed, start failed, crash, app freeze, UI lags, black screen, network error* and *other problems*. Among them, UI glitch and game crash are the two largest categories with UI glitch accounting for 39.95% and game crash occupying 28.76% of compatibility issues. The detailed compatibility issue distribution is shown in Figure 2a. Particularly, UI glitch occurs and has been the most severe compatibility issue in nearly every tested mobile game. TestBird also investigated the proportion of game engines on mobile game applications as shown in Figure 2b, among which Unity3d (also short for Unity) is the most prevalent one and hence our following study is based on the Unity game engine. Another game company WeTest [9] tested all Tencent [7] mobile games and summarized the compatibility issues into eight categories among which UI glitch is also the largest issue and accounts for 47.5% of all the problems. Specifically, the percentage of UI glitches increased by 11% compared to last year and among them the proportion of problems such as *abnormal color block* and *random noise* has increased.



(a) Game issue proportion **(b) Game engine proportion**

Figure 2: Test report from TestBird.

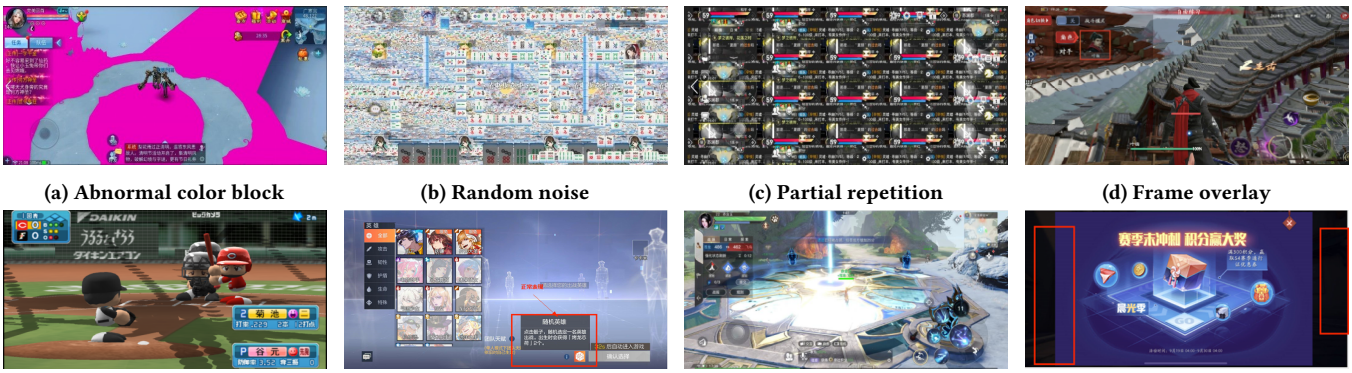
3 CHARACTERISTIC STUDY

Before we build a model to detect game UI glitches, we collected quantities of UI glitch issues that appeared in the real-world game apps. Our study aims to answer the following two questions:

- **RQ1**: What is the general manifestation of UI graphical glitches in mobile game apps?
- **RQ2**: What are the bug causes of these Game UI glitches?

3.1 Data Collection

To better understand the UI glitch issues in real-world mobile game apps, we collect the 466 bug reports of 20 NetEase [4] Android



(a) Abnormal color block (b) Random noise (c) Partial repetition (d) Frame overlay

(e) Object missing (f) Abnormal text (g) Overexposed (h) Black border

Figure 3: Examples of eight categories of game UI glitches.

games belonging to different categories such as Adventure Game, Action Game, First-Person Shooter Game, Role Playing Game, etc. with 2,418 UI glitch images. The main reason we focus on mobile game UI display issues is that compatibility issues between software and hardware frequently appear on mobile devices. Take Android as an example, nowadays more than 10 major versions of the Android operating system (OS) run on 24,000+ distinct device models with different screen resolutions [40]. Because most of the abnormal images are photos captured from the external camera with some annotation rather than the screenshots, we preprocess these images by remaining the screenshots and excluding photos and images with system-related bugs. Finally, we obtain 201 filtered graphical glitch screenshots and use them for our characteristic study.

3.2 Manifestation of Game UI Glitches (RQ1)

Given those collected screenshots, we found that UI glitches are actually versatile in terms of their manifestation and that different types of UI glitches may appear with different frequencies thus has a different level of impact on the game usability. Therefore, categorization of these issues would facilitate our study, design, and evaluation of the related approach. Following the Card Sorting [38] and adapt the technique to the game scenario, we categorize these bugs into 8 categories, namely, *abnormal color block, random pixel noise, partial repetition, frame overlay, model missing, abnormal text, overexposed* and *black border*, and statistic their proportion of occurrence with details as follows:

Abnormal color block (56%). As shown in Figure 3a, abnormal color blocks stretch and cover the UI graph. The main cause is that some material is missing or the camera responsible for rendering pixel RGB values is incorrectly turned off.

Random noise (17%). As shown in Figure 3b, quantities of color pixels randomly distribute over the whole screen or specific area. The main cause is that the camera is incorrectly turned off.

Partial repetition (12%). As shown in Figure 3c, part of the UI area is repeated or mirrored. Disabling camera or post-processing error (GPU does not support the rendering effect or incorrect render logic) may result in this glitch issue.

Frame overlay (6%). As shown in Figure 3d, the frame in the previous time step overlaps the current frame. The wrong value of the camera's clearflag might be the main reason.

Object missing (3%). As shown in Figure 3e, the UI model lacks part of its component. This may be caused by the incorrect values of the alpha channel in the model texture.

Abnormal text (2%). As shown in Figure 3f, multiple pieces of texts are located in the wrong area and may cover the characters or other objects. The main reason for this glitch issue is that the UI is not adapted to the screen resolution.

Overexposed (2%). As shown in Figure 3g, the whole (or part of) UI scene is too bright or overexposed. The main reason is that the intermediate result is stored in a low precision variable and the result is clipped or overflows.

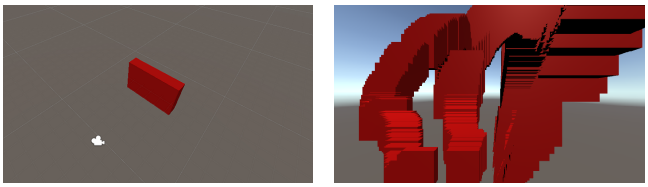
Black border (2%). As shown in Figure 3h, the UI image is not flattened to cover the whole screen and leaves the black borders on both sides because the display resolution (*e.g.*, 854×480) and aspect ratio of some special devices are not considered by the developer.

To ensure the completeness of our study, we asked several game testing experts from the company's development teams to confirm that our summarized UI glitches cover all the common UI issues in their games including not only mobile apps but also other platforms such as PC and PS4. We also demonstrate that our GLIB can be applied to various types of games on distinct platforms and precisely detect UI bug issues in RQ4.

3.3 Bug Causes of Game UI Glitches (RQ2)

After we collect quantities of game UI glitch samples and have a common sense of UI display issues and their threat to the user's game experience, a more important thing is to understand the bug causes of these glitch issues. To common sense, the reason for game UI display issues might be the defects of hardware (*e.g.*, GPU-related issues) or the wrong setting of rendering special effects. To facilitate the visual understanding in detecting UI display issues, we focus on explaining the root causes in terms of source-code level (bugs). By doing so, we collect the historical commit diff of various game apps and categorize the bug issues into 4 major types. For each bug fix example, the code marked with green color is the missing part of the original bug code.

Figure 4: Effect of incorrect camera turned off.



(a) Rendering components **(b) Rendering effect**

Figure 5: Frame overlay generation process.

Rendering cameras are turned off incorrectly. Cameras in the Unity engine are the devices that capture the color and depth information of the game world and display the whole scene to the players. A game scene can hold an unlimited number of cameras, with different objects probably rendered by different cameras. If one camera is turned off unexpectedly, the render results may be substituted by any memory block which has not been initialized, thus the RGB values of the corresponding objects in the image can be randomized, and the manifestation of these random pixel values in game UI display issues will most likely be *abnormal color block*, *random noise* or *object missing*. The bug fix procedure of camera enabled error is shown in Listing 1.

```
  targetCamera.targetTexture = originRT;
+ targetCamera.enabled = true;
  if (UIManager.inst != null && UIManager.inst.uiCamera
      != null)
  {
    UIManager.inst.uiCamera.enabled = true;
  }
```

Listing 1: Bug fix procedure of camera enable error.

Wrong settings of camera's clearflag. Cameras in the Unity engine typically clear the color and depth information on the screen before rendering image frames, and the clearflag function of a camera determines how the color buffer and depth buffer are cleared. If the clearflag instruction is modified incorrectly, the depth and color settings of the scene may get chaotic, *e.g.,* if there is a cube moving randomly in the scene with the blue and gray parts as the background (*i.e.,* the depth is infinite), and the camera's clearflag is incorrectly set as "*Don't Clear*", the color and depth buffers of the previous frame will remain and cause frame repeatedly appear at each time step. The white and red objects in Figure 5a are camera and cube to be rendered, respectively, and the rendering result is in Figure 5b. This bug is regarded as the main cause of *frame overlay*. We fix this bug by setting the camera's clearflag according to the depth buffer. The bug fix procedure of camera clearflag error is shown in Listing 2.

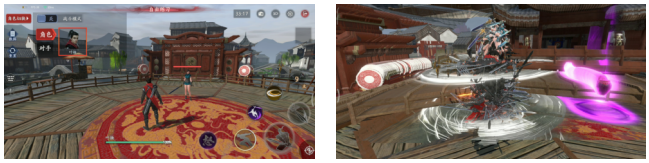


Figure 6: Effect of wrong settings of camera clearflag.

```
  var originRT = targetCamera.targetTexture;
  targetCamera.targetTexture = rt;
  targetCamera.Render();
  CameraPostEffect.instance.doPostEffects(rt, postRT);
  if (UIManager.inst != null && UIManager.inst.uiCamera
      != null)
  {
    if (uiCameraOn)
    {
      UIManager.inst.uiCamera.Render();
      var originUIRT = UIManager.inst.uiCamera.
     targetTexture;
      UIManager.inst.uiCamera.targetTexture = postRT;
      UIManager.inst.uiCamera.Render();
+     UIManager.inst.uiCamera.clearFlags =
     CameraClearFlags.Depth;
      UIManager.inst.uiCamera.targetTexture =
     originUIRT;
    }
  }
```

Listing 2: Bug fix procedure of camera clearflag error.

Post-processing special effects of the previous scene are not cleared in time. Adding post-processing can apply various kinds of filters or effects to the camera's image buffer before an image is displayed on the screen, and this post-processing technique drastically improves the visual expression of the scene. But if the post-processing effect is added incorrectly or if the effect is not cleared in time when the scene changes, the image content will become scrambled, even mess up the whole scene. For example, when one enters a scene without any post-process effects as in Figure 7a, it looks like a man is standing on the ground. However, if a game developer adds a mirror effect to the previous scene (*e.g.,* a lake scene in Figure 7b), and steps into the scene without clearing the post-process effect in time, the image then becomes symmetric as shown in Figure 9c as if there is a lake in the scene. This post-processing special effect bug is likely to cause *partial repetition* issue and the bug fix code is shown in Listing 3.

```
  private static LuaFunction m_hookOnDisable = null;
  private void OnDisable() {
  if (m_hookOnDisable != null) { if(GameBaseObject.Inst
     .InvokeNewHook(m_hookOnDisable, this)) return; }
    if (CameraPostEffect.instance == null || image.
     texture == null)
    {
      return;
    }
    CameraPostEffect.instance.ReleaseUIBloomTarget();
+   CameraPostEffect.instance.ClearPostRenderRT();
  }
```

Listing 3: Bug fix procedure of incorrect camera post-processing effect error.

(a) Raw scene (b) Mirror effect (c) Synthetic scene

Figure 7: Effect of post-processing error.

GPU-related rendering bugs. Some UI glitches are caused by GPU driver bugs or GPU-related rendering bugs. For instance, the version of the operating system (*e.g.,* Android, iOS) on the mobile device might be too old and its handling palettes on the GPU cause UI display issues or the wrong GPU rendering settings like skipping some buffering effect for faster program running might cause *object missing* issue. Wrong GPU rendering settings may also lead to resolution adaption problem and *text out of position* issue.

4 MAIN APPROACH

Our main approach GLIB consists of three parts, first, we propose a source-code based augmentation approach for generating quantities of abnormal game UI display images, then we design a CNN-based image recognition model to learn the pattern of various categories of game UI glitch issues and detect those screenshots with UI display issues, finally, we come up with a saliency map for automatic problem localization. Our GLIB frame is shown in Figure 8.

4.1 Code-Based Data Augmentation

Training a powerful CNN model for visual recognition and UI issue detection requires quantities of data samples. For example, DenseNet [23, 24] uses 50,000 samples from CIFAR [25], 73,257 images from SVHN [33] and 1.2 million images from ImageNet [22] for training. Similarly, our proposed CNN model for UI glitch detection requires a large number of screenshots with versatile UI glitches. Nevertheless, our collected real-world game UI screenshots contain a small proportion of glitch images which also do not fully cover diverse categories of game UI glitches as we mentioned in Section 3.2. Therefore, we propose a code-based data augmentation approach based on the root causes we study in Section 3.3 for generating UI glitch problems by modifying the source code of various mobile game apps and making screenshots for typical scenes.

Particularly, when we inject the corresponding bug code into mobile game execution programs to force UI glitches to occur so that we can collect quantities of screenshots with UI display issues, we must ensure that only UI-related issues happen and other functions of the game apps are not affected (*e.g.,* do not crash) after their programs get updated. We wrap the bug code with execution parameter settings (which we also refer to as patch code) so that the execution program could get updated to the bug-injected version, and this patching technique is called *hotfix*.

Our code-based UI glitch generation approach is automated and can be well-generalized to other games. With *hotfix*, we only need to download the patch code that is pushed to the execution server and the execution programs will get updated automatically rather than reinstalling and recompiling the game apps. Therefore, most mobile apps support *hotfix* for fixing code-related bugs. Particularly, for Unity games, we can insert bug code with the help of *hotfix* without authority and knowledge of the source programs. We inject bug code by changing certain global variables or executing global functions from the Unity native interface. In this way, our code injection approach can be generalized to all Unity engine-based apps and even be easily adapted to graphics engines other than Unity by modifying the global variables and functions in the corresponding engines. Figure 9 illustrate examples of augmented screenshots with UI glitch issues in which the first row shows the normal UI scenes and the second row displays the generated UI glitch scenes.

To be mentioned that GPU-related rendering bugs are typically bottom-layer issues or hardware setting problems and may vary differently from each game, also the manifestation of UI glitches caused by GPU-related rendering bugs is versatile depending on the device itself. Hence, we put this root cause in the future work, and our approach focuses on generating UI glitch scenes with the following three categories.

Turn off cameras in the scene. As we discussed before, the camera is used to capture the objects in the scene, we hence disable some cameras to force the UI glitches to appear in the game apps, we save the screenshots in different scenes as the abnormal samples. The patch code is shown in Listing 4.

```
1 local cameraobjs =
      CS.UnityEngine.Object.FindObjectsOfType(typeof(
      CS.UnityEngine.Camera))
2 for i = 0, cameraobjs.Length-1
3 do
4     cameraobjs[i].enabled=false
5 end
```

Listing 4: Lua patch for turning off all cameras.

Modify camera's clearflag. Camera Clearflag, an *enum* type, determines how to clear the depth buffer and color buffer before rendering the scene. There are four pre-defined values for clearflag: *SkyBox*, *SolidColor*, *DepthOnly*, and Nothing. a) *SkyBox*: clear the color buffer and depth buffer with *SkyBox*; b) *SolidColor*: clear the color buffer and depth buffer with *SolidColor*; c) *DepthOnly*: only clear the depth buffer; d) Nothing: don't clear either color buffer or depth buffer. When we enter a scene, we traverse the camera, set the clearflag as one of the enum values we list above, and check the UI display state, if UI glitches occur, we save the screenshot as an abnormal example. The patch code is shown in Listing 5.

```
1 local  cameraobjs =
      CS.UnityEngine.Object.FindObjectsOfType(typeof(
      CS.UnityEngine.Camera))
2 for i = 0, cameraobjs.Length-1
3 do
4     cameraobjs[i].clearFlags=
      UnityEngine.CameraClearFlags.DepthOnly
5 end
```

Listing 5: Lua patch for changing clearflag as DepthOnly for all existing camera in the scene.

Add incorrect post-processing effect. Adding post-processing effects is the last step in the Unity render pipeline, and the effects can modify the scene style easily. HDR and background blurring are two common post-processing effects. The UI image frame could become scrambled if we add incorrect post-processing effects to the scene (*e.g.,* adding the mirror effect to the button can lead to

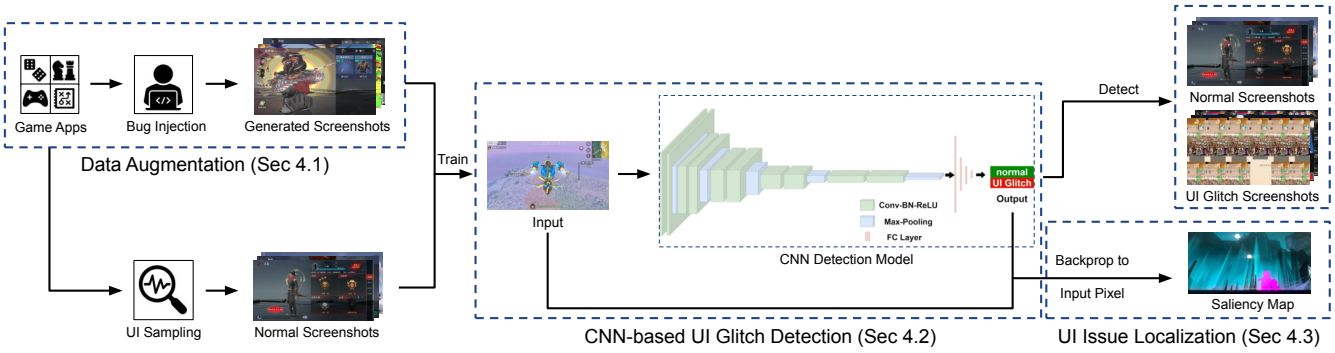


Figure 8: Overview of GLIB.

(a) Camera enabled **(b) Camera clearflag** **(c) Post-processing**

Figure 9: Examples of code-based data augmentation.

partial repetition). We randomly choose some post-processing effects and add them to different scenes and save the screenshot as the abnormal sample if the UI scene is messed up. The patch code is shown in Listing 6.

```
local detectCamera = GameObject.Find("UICamera")
if (detectCamera ~= nil)
then
    detectCamera.gameObject: AddComponent(typeof(
     CameraFilterPack_3D_Mirror))
end
```

Listing 6: Lua patch example for adding mirror post-processing effect to current scene.

Note that there are two main reasons why we do not directly apply our summarized code patches to find bugs in game source code. First, injecting a type of bug requires to know only one code pattern of such bug type but detecting bugs requires knowing all patterns of this type of bug. In our patch, we only need to change some global variables or functions to generate UI glitches. However, for each bug type, there may be thousands of relevant statements and the correctness of each statement depends on other context codes. Second, even if one can figure out a code-analysis DL model for detecting bugs, the source code of the application under test is not always available (noted that our bug injection needs access to the source code of the training applications only). Thus, our GLIB is a more general and effective approach for real-world testing cases.

4.2 CNN-Based UI Glitch Detection Model

Deep Learning has achieved remarkable success in computer vision tasks such as image classification, object detection, object tracking, etc. and we hence choose the CNN architecture for detecting abnormal UI display images which can be regarded as one kind of image classification tasks.

Given the screenshot as input to our CNN model, we firstly resize the input to a fixed size whose width and height is $w \times h$, then we use convolutional kernels to extract feature maps of the input followed by pooling layer which can progressively reduce the spatial size of feature representation meanwhile control overfitting. To improve the stability of CNN, the Batch Normalization (BN) is added after each convolutional layer. We obtain feature maps from the last convolutional layer and send them to the multiple full-connection (FC) layers to train a classifier with the K-dimensional vector as the output. Finally, the probability distribution of each class c is computed by softmax function:

$$P(y = c|x) = \frac{e^{f_c(x)}}{\sum_{k=1}^{K} e^{f_k(x)}} \tag{1}$$

The classification result is given by the argmax function:

$$label = \arg\max_{c} P(y = c|x). \tag{2}$$

To increase the nonlinearity of the CNN model, an activation function is added after the BN layer and FC layer.

4.3 Saliency Map

The CNN model only determines whether the image is abnormal, however, we are more concerned about which part of an image is abnormal thus can help the developer to fix the bug. Moreover, the saliency map [37] can help to understand whether the model is accurate and how the model works. We simply compute the derivative of the label to the input by

$$\frac{\partial f(I)}{\partial I} \tag{3}$$

where f represents the CNN model and I is the input image. The bigger gradient value indicates a larger contribution to the classification result. If the output shows that the image is abnormal, the pixel with a large gradient value indicates the abnormal area.

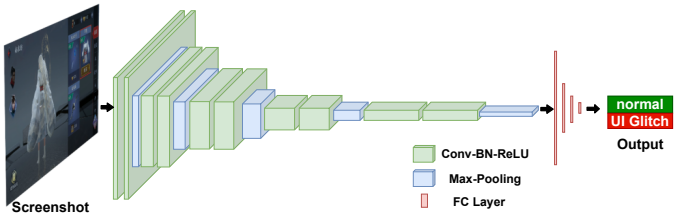


Figure 10: The architecture of CNN.

4.4 Implementation

We resize the input to a fixed size 512 × 256, if the image is vertical, we will rotate it to horizontal then resize it to the fixed size. Our CNN model consists of 10 convolutional layers whose kernel size is 3 × 3 followed by batch normalization layers, 5 MaxPooling layers, and 4 fully connected (FC) layers with the output size of the last layer $K = 2$, the activation function we adopt is ReLU. We set the number of kernel as 16 for convolutional layer $1 \sim 4$, 32 for layer $5 \sim 6$, 64 for layer $7 \sim 8$ and 128 for layer $9 \sim 10$. The model updates its parameter by minimizing the CrossEntropy loss for the two-label classification task. The network is trained by Adam optimizer over batches of 16 input images with an initial learning rate λ as 0.001. The detail of our ConvNet configurations is shown in Figure 10. We implement our model based on the Pytorch framework.

5 EXPERIMENT DESIGN

Our experiment is designed to answer the following questions:

- **RQ3:** How effective is GLIB in terms of detecting game UI glitches?
- **RQ4:** How well does GLIB perform in real-world game applications?

5.1 Experimental Setup

The game UI screenshots in our experiment are composed of 2 parts: screenshots without UI glitches (normal images) and screenshots generated by code-based augmentation approach (glitch images). We combine the two parts of data samples to train our GLIB.

To balance the distribution of our training data, we roughly set the number of normal images and that of glitch images as 1: 1, particularly, we randomly select 6,841 screenshots from all the code-based generated glitch images for training, among which 2,511 of them are generated by setting the wrong clearflag of the scene camera, 3,763 of them are generated by turning the camera off and 567 of them are generated by adding incorrect post-processing effect. There are 6,817 normal screenshots and 6,817 glitch screenshots in the training dataset, 783 normal screenshots, and 759 glitch images in the validation dataset where 278 of them are generated by setting the incorrect camera clearflag, 418 are generated by turning the camera off and 63 are generated by adding incorrect post-processing effect.

The screenshots are collected from two games, we manually traverse the scene in the mobile game app by clicking randomly on the mobile screen, capture the screenshots until UI components are stable, and save the screenshots as bug-free data. Then we apply three patches in Section 4.1 to the game, if the screen is blurred, we capture the screen as the code augmentation result.

Table 1: Data Distribution

Data Type	Augmentation Approach(s)	Game1	Game2
Normal		1654	6186
Glitch		47	85
Rule	Partial Repetition	1654	6186
	Solid Color Block	1654	6186
	Mosaic Effect	1654	6186
	Random Noise	1654	6186
Code	Camera Turned Off	1506	3056
	Incorrect Camera Clearflag	3144	1076
	Incorrect Post-Processing	330	300

Note that game scenes are typically dynamic rather than static. In each scene there may be multiple moving UI objects which produce a different screenshot in the next frame, thus there are quantities of different screenshots in each scene. Given that each game contains abundant different scenes, we can produce sufficient diverse abnormal screenshots for well-fitting the model. The rule-based data augmentation is an offline approach thus is processed after all the bug-free screenshots are collected. Table 1 shows the distribution of screenshots we collected.

The test dataset that we use to evaluate the model is collected from 466 historical bug reports. We exclude the screenshots of game1 and game2 as well as some low-quality images and finally get 192 glitch images.

Table 2: Experiment Setup

Approach	Glitch Image	Normal Image	Total
Base	107 / 25	6817 / 300	6924 / 325
Rule	6817 / 783	6817 / 783	13634 / 1566
Code	6841 / 759	6817 / 783	13658 / 1542
Code+Rule	13658 / 1542	13634 / 1566	27292 / 3108

5.2 Baselines

To further demonstrate the advantage of our proposed data augmentation approach, we compare GLIB with five baselines utilizing deep learning techniques to examine the UI glitch detection effect. Because our goal is to detect UI glitches via bug understanding, for all the baseline we use the same CNN model and only with different data handling techniques. The dataset size of each baseline is listed in Table 2.

Base. We search the historical test reports of game1 as well as game2 and collect 132 screenshots which are truly bug images confirmed by the development teams. We select 125 of the 132 glitch screenshots and combine them with 6,817 normal images that we collect from game1 and game2 to build the training dataset without any augmentation approach, and our evaluation dataset consists of left 25 glitches screenshots and 300 normal images. We exclude the game1 and game2 screenshots from the 201 filtered graphical glitch screenshots which are collected from 20 game apps and remain 192 glitch screenshots for the test procedure.

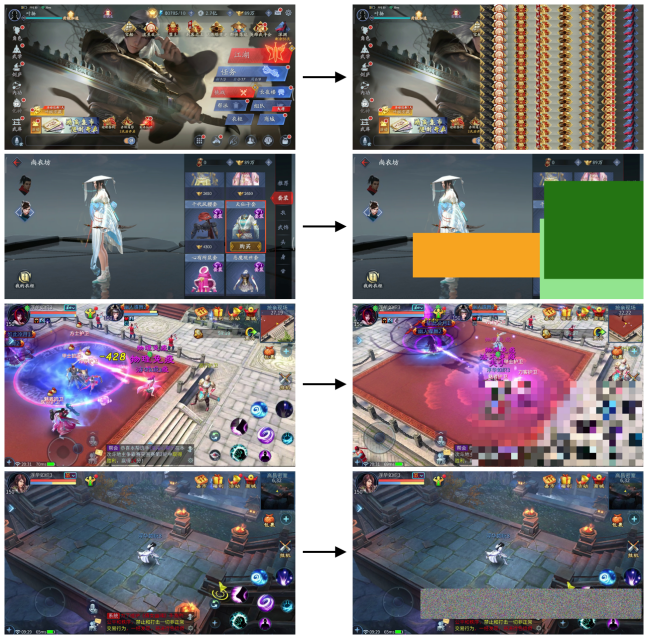


Figure 11: Examples of rule-based data augmentation, the four rows from top to bottom correspond to image partial repetition, adding solid color block, adding mosaic effect and adding random noise, respectively.

Rule(R). The heuristic-based data augmentation approach proposed by `Owl Eyes` [29] contains several rules to approximate the UI display issues in non-game apps. Because most of their rules are based on text-relevant UI bugs (*e.g.,* NULL value, text overlap) which rarely appear in a game scenario, we adapt their rules to our studied manifestation of game UI glitches – for each of the UI display issues, we generate screenshots by randomly choosing one of the following four rules. 1) *Image partial repetition*: we randomly choose a rectangle area in an image, then repeat sampling in a horizontal or vertical direction; 2) *Adding solid color block*: we generate 3 ∼ 5 blocks where all pixels share the same color in one block and put them on an image randomly one by one, thus the former color block may be partial covered by the latter one. Color of each block can be arbitrary RGB value; 3) *Adding mosaic effect*: we randomly choose a rectangle area in an image, dividing the area into several small patches where every pixel has the same RGB value as the center pixel in one patch; 4) *Adding random noise*: we randomly choose a rectangle area in an image and set RGB value randomly for every pixel in the rectangle. The four pairs of rule-based generated screenshots are shown in Figure 11.

We generate abnormal samples based on the normal data with four simple heuristic approaches we discuss above, note that each normal screenshot is used only once. The final training data contains 6,817 normal screenshots and 6,817 generated glitch screenshots where 1,701 of them are generated by *image partial repetition*, 1,659 of them are generated by *adding solid color block*, 1,752 of them are generated by *adding random noise* and the left 1705 of them are generated by *adding mosaic effect* to the screenshots. The 1,566 evaluation samples are composed of 189 partial repetition glitch images, 184 abnormal color block screenshots, 220 random noise screenshots, 190 mosaic effect screenshots, and 783 normal screenshots.

Rule(F). For the second rule –*adding solid color block*–we choose the RGB value of generated color blocks from the pre-defined four color values (red, black, pink and cyan) rather than arbitrary value due to the prior knowledge that these 4 color appears mostly when the material of game objects is missing or settled during the loading process. The other setting in this baseline is the same as Rule(R) approach.

Code+Rule(R). We combine both code-based and rule(R)-based generated screenshots as the training dataset for modeling the UI glitches.

Code+Rule(F). We combine both code-based and rule(F)-based generated screenshots as the training dataset for modeling the UI glitches.

5.3 Evaluation Metrics

To evaluate the overall effectiveness of our proposed game UI display issue detection approach, we apply four commonly used evaluation metrics in image classification tasks, *i.e.,* accuracy, precision, recall, F1-score [30, 36]. For all the metrics, a higher value indicates better model performance.

Accuracy. Accuracy reflects the trained model's ability to make correct decisions on the test set. The more correct samples the model predicts, the higher accuracy it will output.

Precision. Precision presents the proportion of correctly classified screenshots as UI glitch among all screenshots predicted as UI glitch.

Recall. Recall indicates the proportion of correctly classified screenshots as UI glitches among all screenshots that have UI display issues.

F1-score (F-measure). F1-score is calculated from the precision and recall of the test and it reflects the harmonic mean of precision and recall. The highest possible value of an F1-score is 1 which indicates perfect precision and recall, and the lowest possible value is 0 if either precision or recall is zero.

6 RESULTS AND ANALYSIS

6.1 UI Glitch Detection Performance (RQ3)

We evaluate the effectiveness of our `GLIB` and the baseline approaches on the testing dataset composed of the collected 192 abnormal screenshots with UI glitches and 365 normal screenshots, the experiment results are listed in Table 3. We can see that our code-based data augmentation approach achieves the overall best performance (*i.e.,* highest precision/recall/F1_score/accuracy). The 76.7% increment of recall compared to the base approach indicate the effectiveness of our code-based data augmentation approach. We search for only one false negative sample and find that the error area in this screenshot is too tiny to be recognized even for humans.

Particularly, the five baseline approaches all achieve high precision, indicating that most screenshots predicted by the model as abnormal have UI glitches. The base approach trained without any augmentation approach by only using the original glitch screenshots has the lowest recall (56.3%), indicating that almost half of

the UI glitch images are incorrectly classified by the model if not sufficient UI glitch samples are learned. We search the classification result and find that the model cannot detect the screenshots with UI glitches such as partial repetition, abnormal text, and abnormal color block. Though these categories occupy a large proportion of UI glitch issues, the number of glitch images is too small to cover the different patterns of these categories, and hence the model cannot learn the UI glitch manifestation sufficiently.

Table 3: Experiment Results

Approach	Precision	Recall	F1_score	Accuracy
Base	0.991	0.563	0.718	0.803
Rule(R)	0.935	0.677	0.785	0.836
Rule(F)	0.974	0.594	0.738	0.813
Code+Rule(R)	0.990	0.984	0.987	0.988
Code+Rule(F)	0.995	0.948	0.971	0.975
GLIB	**1.000**	**0.99**	**0.997**	**0.998**

Rule(F) incorrectly classifies 78 glitch screenshots as normal. We check these images and find that the glitch issues of them are mainly abnormal color block and text overlap. As we mentioned before, we only adopt four colors (black/red/pink/cyan) to generate the Solid Color Block in Rule(F) approach, which may cause the detection failure when a new color block appears. We check the bug reports and find that color blocks other than the pre-defined four types appear due to material missing. However, this unexpected error only occurs on a special GPU that has a special order of RGB values, which indicates that the manifestation of UI glitches on different devices caused by the same bug can still be different. The performance of Rule(F) is degraded by text overlap glitch issues because we do not consider the rules of abnormal text as we cannot localize the text area in UI screenshots without labeled JSON files that are typically not supplied by game engines. Moreover, in game apps some texts are displayed as Word-Arts or images but not the order of standard characters, thus the localization technique that uses OCR tools cannot work.

We study the 62 false-negative samples from Rule(R) and find that the main UI glitch categories of them are text overlap and abnormal color block, particularly, these color blocks are transparent and can be easily recognized as part of the background object. This transparent color block is similar to the dialog box in games and is misclassified also because we didn't generate the text-relevant glitch images in our training dataset. Because the transparent color blocks do not appear in the training dataset of the Rule approach, it is straightforward that the model cannot this type of UI glitches. Moreover, we find that the glitch images with blue color blocks are detected as UI glitch images whereas Rule(F) regards them as normal images, the reason may be that the Rule(R) approach can generate blue color blocks that can't be produced by Rule(F).

For Code+Rule(R) and Code+Rule(F), their recalls are largely improved compared to the single Rule(R) and Rule(F) approach, which demonstrates that the glitch images generated by our code-based approach can facilitate the model to learn more effectively. However, the reason that the combined approaches are not as effective as the single code-based augmentation approach may be that the distribution of code-based generated samples and rule-based generated samples are not identically consistent which may affect the training result.

6.2 Practical Evaluation (RQ4)

To examine the practical value of our GLIB, we collect two PC games from the official website, three iOS games from App Store [3] and nine Android games from TapTap [6] development teams. These games are developed by different game engines and none of these apps appear in the training dataset.

Airtest [1] is a cross-platform UI automatic game testing framework, and testers can write test scripts in Airtest IDE to execute specific test cases in the mobile device. Airtest IDE also provides a screen capture API for testers to take screenshots when necessary. We use the screen capture API to collect screenshots of various kinds of UI events (*e.g.*, click, swipe, long press, etc.) from the 14 games by running different test cases. We generate in total 2,100 screenshots from the 14 games, an average of 150 screenshots are obtained for each app. We then feed those screenshots to our GLIB for detecting abnormal UI issues. Once a UI glitch is spotted, we record the bug and report the issue to the app development team.

Table 4: Detected Game Issues

Game Name	Game Category	Source	Daily Active Users	Download
Justice	Role-Playing	Official Web	300K+	50M+
A Chinese Ghost Story	Role-Playing	Official Web	300K+	50M+
Ghost	Role-Playing	TapTap	700K+	100M+
Revelation Mobile	Role-Playing	TapTap	500K+	10M+
Love is Justice	Love	TapTap	50K+	5M+
UNO	Card	App Store	50K+	5M+
Fever Basketball	MOBA	App Store	5K+	5M+
Marvel Duel	Card	TapTap	5K+	500K+
Oracle Civilization	Simulation	App Store	1k+	100k+
Ghost World Chronicle	Card	Develop Team	N/A	N/A
Elysium Of Legends	Card	Develop Team	N/A	N/A
phase10	Card	Develop Team	N/A	N/A
Fpus	Shooting	Develop Team	N/A	N/A
The Absolute Acting	Simulation	Develop Team	N/A	N/A

Table 5: Practical Evaluation Results

Platform	GLIB	Rule(R)	Rule(F)
PC	7	3	1
Android	35	22	15
iOS	11	6	5
Total	53 (48 confirmed)	31 (28 confirmed)	21 (17 confirmed)

Table 4 lists all bug issues detected by our GLIB and Table 5 shows the number of detected UI glitch issues on different platforms by the three approaches. In sum, GLIB detects 53 glitch issues and 48 of them are confirmed and fixed by the game development team; Rule(R) spots 31 glitch issues and 28 of them are confirmed and fixed; Rule(F) detects 21 glitch issues and 17 of them are confirmed and fixed. These confirmed and fixed bugs further demonstrate the effectiveness of the practical value of our proposed approach in detecting game UI glitches.

7 CASE STUDY

To demonstrate that GLIB can accurately localize the glitch area of detected abnormal screenshots, we apply the saliency map introduced in Section 4.3 to show developers more detail about our model's prediction. We randomly select some images that are classified by GLIB as glitch images and calculate the derivative of the output concerning each pixel in the input image, the results are shown in Figure 12 where the original screenshots are placed in the left and the generated saliency maps (which are converted to heat-maps) are listed in the right. A brighter area in the heat maps indicates a larger gradient of corresponding pixels, i.e, these pixels contribute more during the classify progress and are more likely to be the UI glitch issues. The UI glitch of the first image in Figure 12 is partial repetition, and the corresponding saliency map shows that GLIB concentrates much on these repetition areas, which is consistent with the manifestation. The saliency maps of the second and the third images indicate that the abnormal color blocks rather than the other background elements are the buggy area of the screenshots, which also agrees with the manifestation. From the saliency map of the glitch screenshots, we can see that the model can not only accurately detect the image with UI glitch issues, but can also localize which part of the screenshot is abnormal.

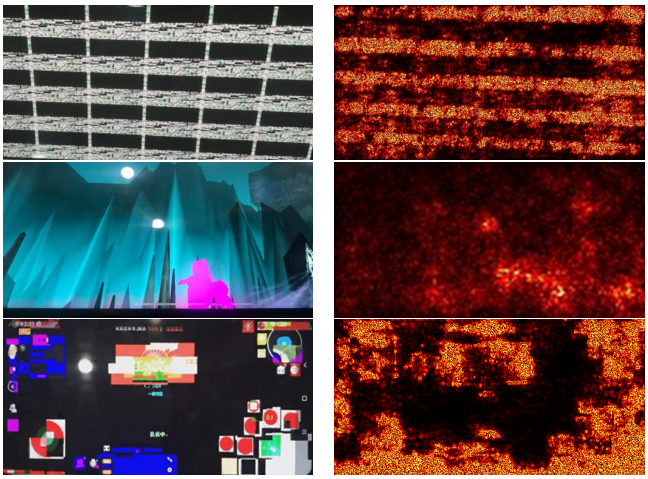


Figure 12: The images in the left column are the glitch images and images in the right column are the corresponding saliency map, the red pixel contribute most to the final label.

8 DISCUSSION

In this section, we discuss the generality of our approach.

Generality across games. Our training data are collected and augmented from 2 Acting games, which may limit the model's applicability in real-world practice. However, our testing data used in RQ3 consists of 20 games including categories such as Role-playing games, card games, shooting games, MOBA (Multiplayer Online Battle Arena), love games, simulation games, etc. which nearly cover all popular daily-used game apps. The evaluation result shows that our proposed GLIB can accurately detect (99.5% recall) all screenshots with UI glitches. This further demonstrates the generality of GLIB across different types of games. Moreover, our GLIB is a black-box image augmentation approach that requires source code only in the training phase and the well-trained model can be directly used to detect UI glitches on other games.

Generality across languages. Another advantage of our GLIB is that it can be applied for detecting UI glitches on game applications with different languages. Although the testing data of our experiment for RQ3 and case study only contains the screenshots of Chinese games, our study in Section 3.2 shows that most of the game UI glitches are text-irrelevant, *i.e.,* abnormal text-only account for 2% of all the UI display issues, plus our code-based data augmentation approach mainly focuses on UI glitches that are language-invariant such as abnormal color block, random noise, partial repetition and frame overlay caused by rendering effect or post-processing effect error. Hence, our proposed GLIB can be generalized for UI glitch detection in games with other languages.

Generality across platforms. Even though different games may run on different platforms, the game UI content is mostly decided by novel images which consist of 3d models as well as UI components. To prove that our model can precisely detect glitch images in terms of different game platforms, we collect UI screenshots from 9 Android games, 3 iOS games, and 2 PC games and use them as our testing dataset. The experiment results in RQ3 show that our approach can accurately detect all UI glitch images from games with different platforms, which further demonstrates the feasibility of our proposed GLIB.

Generality across engines. The game engine is a software development environment that provides developers with a series of tools to facilitate easy program writing. Games based on different engines may have different program structures, but the game UI rendering effect mainly depends on the low-level CPU/GPU system calls. For the same root cause (bug error), the manifestation of glitch images is typically similar regardless of which game engine the game is based on. To prove that our GLIB can well recognize glitch images across different engines, we select 2 games developed by a self-defined engine, 1 unreal-engine-based game, and 11 Unity games to compose our test dataset in RQ3. The experiment results show that our model can be generated well across different game engines.

9 RELATED WORK

Our work, inspired by the automatic GUI testing [10, 31, 39] combined with deep learning technique, proposes a game GUI bug detection approach. GUI, a visual interface connecting users and software programs, has been studied by many researchers on different topics. Automatic GUI testing dynamically explores GUIs of an application, and several approaches [26, 42] use computer vision techniques to detect GUI components to make predictions and compare different tools for GUI testing on Android applications. Recent deep learning-based techniques [20, 41] have also been applied for automatic GUI testing. More work on GUI with computer vision techniques such as GUI search [11–13, 15, 17, 35, 43] and GUI code generation [14, 18, 19, 32, 34] facilitates the effective completion of computing tasks based on image features.

On the other hand, many software linting tools aiming to flag bugs, stylistic errors, programming errors, and suspicious constructs [16, 44] have been proposed to ensure the normal operation of GUI. For example, StyleLint [5] helps developers avoid errors and enforce conventions in styles, Android Lint [2] reports over 260 different types of Android bugs including correctness, performance, security, usability, and accessibility. Different from static linting, our GLIB dynamically explores GUIs of an app as what automatic GUI testing does, but note that these GUI testing techniques concentrate on functional testing, whereas our work focuses on non-functional testing (*i.e.,* GUI glitches typically do not cause app crash but negatively affect the app usability). We analyze the GUI display issue in terms of software rendering bugs such as rendering camera settings and post-processing effects error which cause the app compatibility problems. It is extremely difficult and expensive for the developers to cover all the popular contexts when conducting testing. Moreover, our work only requires the screenshots as the input rather than these works based on static or dynamic code analysis. This crucial characteristic makes it easier for our lightweight CNN-based model to learn the pattern of UI glitch images and localize the UI glitches on the screenshot by a saliency map [37] and also makes our approach more generalized to a different platform.

10 THREATS TO VALIDITY

In our GLIB framework, the only manual part is to traverse and identify multiple diverse game scenes in each game for building our original training dataset. Also, our defined three categories of code injection approaches are based on study and empiricism. The code injection data augmentation based on *hotfix* patching technique, CNN-based UI glitch detection as well as UI issue localization are all automated and can be easily adapted to other games on different platforms. Inappropriate selection of game screenshots in the manual part may weaken the *external validity* of experimental conclusions. We try to mitigate this threat by traversing and selecting as many as distinct game scenes to make the dataset diverse and abundant. Our *internal threat* mainly arises from the completeness of manifestation of game UI glitches and our defined three types of code patching approaches. We ask several game developing experts from NetEase to confirm that our summarized eight categories of UI glitches do cover all the common issues in their game apps. We also show that our code injection can generate all the most common five types of UI glitches.

11 CONCLUSION AND FUTURE WORK

Detecting and improving the quality of mobile game applications is of great value for nowadays game developers and testers. This paper proposes an automated UI glitch detection approach based on deep learning and bug analysis. Our empirical study on the root causes of game UI glitches facilitates a code-based data augmentation approach. Experimental results show that our GLIB is effective and shows great advantage in a real-world game testing scenario, *i.e.,* achieving nearly 100% precision, recall, and F-1 score for classifying screenshots collected from 20 popular games, way better than the existing rule-based approach. Also, as the first work on game UI testing, we contribute to a systematical investigation of UI glitches in real-world mobile game apps, as well as a large-scale dataset of game app UIs with display issues for follow-up studies. Our proposed test oracle for automated UI glitch detection could facilitate further study on game UI testing.

In the future, we will focus more on the GPU-related rendering issues that cause game UI glitches and also keep improving the functionality of our model. Specifically, GLIB is one part of the game testing framework that we plan to research in the future. The whole framework contains an *IO module*, a *scene traverse module*, GLIB, and a *log module*. First, *IO module* captures screenshots from a mobile device and feeds them to both GLIB and the *scene traverse module*; Second, GLIB classifies the given screenshot as normal or glitch (*log module* loggings the corresponding information), and the *scene traverse module* recognizes UI and click to yield the next scene, then it chooses a UI and returns the UI back to *IO module*; Last, we repeat the two steps to realize the whole automated game testing. Moreover, we hope to find a tight connection between bugs and the characteristic of UI glitches so that we can predict the bug code given a screenshot with UI display issues. And this bug inference technique can be more valuable when guiding developers to fix app compatibility issues.

ACKNOWLEDGMENTS

We would like to thank Lei Ma, Zihe Song and Simin Chen for their help. We also thank the anonymous reviewers for their helpful feedback. This work was supported in part by UT Dallas startup funding #37030034.

REFERENCES

[1] [n.d.]. Airtest: Automated testing private cloud solution. ([n. d.]). https://airtest.netease.com/home/
[2] [n.d.]. Android lint: Android Studio Project Site. ([n. d.]). http://tools.android.com/tips/lint
[3] [n.d.]. iOS App store - Apple. ([n. d.]). https://www.apple.com/app-store/
[4] [n.d.]. NetEase games: Thunder fire studio. ([n. d.]). https://leihuo.163.com/en/
[5] [n.d.]. Stylelint: A mighty, modern linter that helps you avoid errors and enforce conventions in your styles. ([n. d.]). https://github.com/stylelint/stylelint
[6] [n.d.]. TapTap: Discover superb games. ([n. d.]). https://www.taptap.com/
[7] [n.d.]. Tecent Games: Connecting users in everyday life. ([n. d.]). https://www.tencent.com/en-us
[8] [n.d.]. TestBird: Effeciently connect China and the global market, and continue to empower global enterprises with cutting-edge technologies. ([n. d.]). https://www.testbird.com/
[9] [n.d.]. WeTest. ([n. d.]). https://wetest.qq.com/
[10] 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.
[11] Farnaz Behrang, Steven P Reiss, and Alessandro Orso. 2018. GUIfetch: supporting app design and development through GUI search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 236–246.
[12] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From Lost to Found: Discover Missing UI Design Semantics through Recovering Missing Tags. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW2 (2020), 1–22.
[13] 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.
[14] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI design image to GUI skeleton: a neural machine translator to bootstrap mobile GUI implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.
[15] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI design search through image autoencoder. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–31.

[16] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile GUI components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 322–334.

[17] 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.

[18] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 596–607.

[19] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2019. GUI-squatting attack: Automated generation of Android phishing apps. *IEEE Transactions on Dependable and Secure Computing* (2019).

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

[21] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 845–854.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[23] Gao Huang, Zhuang Liu, Geoff Pleiss, Laurens Van Der Maaten, and Kilian Weinberger. 2019. Convolutional Networks with Dense Connectivity. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019).

[24] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[25] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[26] Tomi Lämsä. 2017. Comparison of GUI testing tools for Android applications. *University of Oulu* (2017).

[27] Yufei Li. 2021. GLIB. (2021). https://doi.org/10.5281/zenodo.5108667

[28] Yufei Li. 2021. GLIB: image dataset. (2021). https://doi.org/10.5281/zenodo.5081242

[29] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 398–409.

[30] Suyu Ma, Zhenchang Xing, Chunyang Chen, Cheng Chen, Lizhen Qu, and Guoqiang Li. 2019. Easy-to-deploy api extraction by multi-level feature embedding and transfer learning. *IEEE Transactions on Software Engineering* (2019).

[31] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 559–570. https://doi.org/10.1145/2884781.2884853

[32] 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.

[33] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning. (2011).

[34] 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.

[35] Steven P Reiss, Yun Miao, and Qi Xin. 2018. Seeking the user interface. *Automated Software Engineering* 25, 1 (2018), 157–193.

[36] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge.

[37] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).

[38] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[39] 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.

[40] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237.

[41] 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.

[42] Samer Zein, Norsaremah Salleh, and John Grundy. 2016. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software* 117 (2016), 334–356.

[43] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: Vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 350–361.

[44] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: vision-based linting of GUI animation effects against design-don't guidelines. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1286–1297.

[45] 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.