Efficiency Matters: Speeding Up Automated Testing with GUI Rendering Inference

Sidong Feng
Monash University
Melbourne, Australia
sidong.feng@monash.edu

Mulong Xie
Australian National University
Canberra, Australia
mulong.xie@anu.edu.au

Chunyang Chen*
Monash University
Melbourne, Australia
chunyang.chen@monash.edu

arXiv:2212.05203v2 [cs.SE] 26 Feb 2023

***Abstract*—Due to the importance of Android app quality assurance, many automated GUI testing tools have been developed. Although the test algorithms have been improved, the impact of GUI rendering has been overlooked. On the one hand, setting a long waiting time to execute events on fully rendered GUIs slows down the testing process. On the other hand, setting a short waiting time will cause the events to execute on partially rendered GUIs, which negatively affects the testing effectiveness. An optimal waiting time should strike a balance between effectiveness and efficiency. We propose AdaT, a lightweight image-based approach to dynamically adjust the inter-event time based on GUI rendering state. Given the real-time streaming on the GUI, AdaT presents a deep learning model to infer the rendering state, and synchronizes with the testing tool to schedule the next event when the GUI is fully rendered. The evaluations demonstrate the accuracy, efficiency, and effectiveness of our approach. We also integrate our approach with the existing automated testing tool to demonstrate the usefulness of AdaT in covering more activities and executing more events on fully rendered GUIs.**

***Index Terms*—Efficient android GUI testing, GUI rendering, Machine Learning**

I. Introduction

GUI (Graphical User Interface) is one of the most common forms of user interface that provides a visual bridge between a software application and end-users through which they can interact with each other. Since most bugs or issues can be spotted by users in GUI, GUI testing is widely used to ensure app quality. There are many automated GUI testing works based on randomness [1], [2], app artifact (e.g., source code, activity) [3], [4], reverse engineering [5], [6], and deep learning [7], [8]. For most dynamic GUI testing tools, the more time for testing, the higher testing coverage, the more likely to find bugs, and the higher quality of the app release. However, due to the budget limit and market pressure, development teams have to meet deadlines by striking a balance between testing time and other demands [9].

Given limited testing time, improving testing efficiency means more test cases, leading to relatively high-quality apps. Towards that target, there have been some works [10], [11], [12] leveraging advanced infrastructure support to fetch GUI hierarchy and execute events efficiently. Besides the infrastructure efficiency, the impact of GUI rendering has been overlooked. GUI rendering is the act of generating a frame from the app and displaying it on the screen [13] including transiting from the last page, loading resources from internet, drawing UI objects (button) into pixels following the order of view hierarchy, etc, as seen in Fig. 1. That process may be long depending on the app code quality, device performance and internet bandwidth. Typically, automated testing tools configure a fixed amount of time (throttle) between events, in order to wait for the GUI to be fully rendered. Setting an optimal throttle can help reduce the waiting time of automated testing tools, resulting in higher testing efficiency.

However, the fixed throttle may not work for different testing tools on different devices, and even different pages in the same app due to the screen complexity difference of each GUI. Although long throttle can bring fully rendered GUIs, it may slow down the whole testing process for some idle waiting (e.g., Fig. 1E). If the throttle is too short, many GUIs may just be partially rendered which negatively affect the testing effectiveness [14], [15], [16], [17] due to two major reasons. First, there are many GUI testing tools highly dependent on visual information of GUI, including usability bug detection [18], robot testing [19], reinforcement-learning-based app exploration [20], test case migration across platforms [21] which require fully rendered GUI as the input. Second, the run-time view hierarchy may be out of sync with the rendering GUI, and the action based on the view hierarchy may not be executed as expected, resulting in low coverage (e.g., tapping "Screws" image by coordinates from the view hierarchy file will be missed in rendering GUI at Fig. 1C). Therefore, an adaptive throttle (e.g., 600ms in Fig. 1) is needed to strike a balance between effectiveness and efficiency.

To further understand the throttling issues in automated testing tools, we first carry out a pilot study on 3 widely-used GUI testing tools for 32 apps to observe the GUI rendering. Results show that a fixed short throttle setting (e.g., 200ms) causes 24% of events on average happening on partially rendered states. The partially rendered states mainly include *transiting state*, *explicit loading state*, and *implicit loading state*. Although extending the throttle interval can help address the issues with a partially rendered state, an excessive long throttle (e.g., 1000ms) reduces 52.8% testing events of automated exploration, which can seriously impact the testing efficiency. These findings motivate this work in finding an adaptive throttle during GUI testing, and the difference between fully rendered and partially rendered lays the foundation

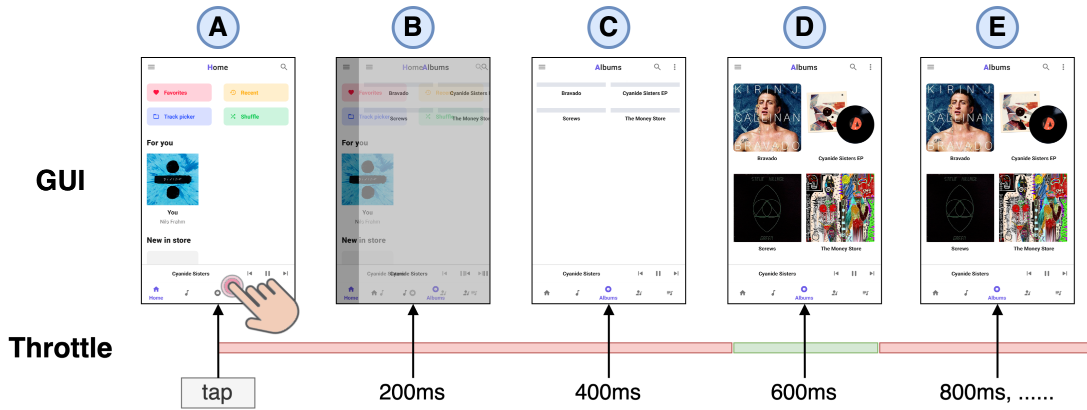


Fig. 1: Automated GUI testing with different throttles. Green bars represent ideal throttle, red bars represent flawed throttles that ineffectively test on partially rendered states or inefficiently stagnate on GUI.

for our approach.

We propose a lightweight approach AdaT, to automatically throttle the events adaptive based on GUI rendering inference. Specifically, we formulate the throttle time prediction problem as a run-time classification task by discriminating between fully rendered and partially rendered GUIs. We adopt a deep learning method to model the visual information from the GUI screenshot for inferring the GUI state. First, we leverage image processing techniques to extract frames from GUI transiting screencasts to construct a large-scale binary GUI dataset, including 66,233 fully rendered and 45,623 partially rendered GUIs. Then, we adopt a small but efficient Convolutional Neural Network (CNN) based approach to discriminate the GUI rendering state. To deploy our approach in testing tools to synchronize GUI rendering inference and schedule testing events, so as to send events until the GUI is fully rendered, we implement a socket-based framework to stream the real-time GUI screenshots and GUI rendering inference. Note that one strength of our approach is that it is purely image-based, which can be easy to deploy in real-world practice.

To evaluate the accuracy of our AdaT, we carry out a large-scale experiment on 20,125 GUI screenshots from 1,877 Android apps. Compared with 11 state-of-the-art baselines, our AdaT can achieve more than 99.8% accuracy in predicting GUI rendering state. We also conduct an experiment to demonstrate that our approach can speed up automated testing without sacrificing testing effectiveness by replaying 18 existing crash bugs from 12 defective Android apps. In addition, the efficiency of current GUI testing tools is further boosted by novel approach design and implementation. Given the same run-time for the testing tool with and without AdaT, the AdaT-enhanced tool can achieve 6.96% higher activity coverage and execute 13.24% more events, than the vanilla tool.

The contributions of this paper are as follows:

- To the best of our knowledge, this is the first study to automatically infer GUI rendering status for accelerating GUI testing. We propose a lightweight computer-vision based approach, AdaT[1] to adaptively adjust the throttle between events.
- A motivational empirical study to understand GUI rendering process in Android apps to motivate this study and lays the foundation for our research.
- Comprehensive experiments including the performance of AdaT and its integration with the automated testing tool to demonstrate the accuracy, efficiency, effectiveness, and usefulness of our approach.

II. Motivational Study

To better understand the issues of automated testing tools with throttling, we carried out a pilot study to examine the prevalence of these issues, so as to facilitate the development of our tool to enhance the existing Android testing tools.

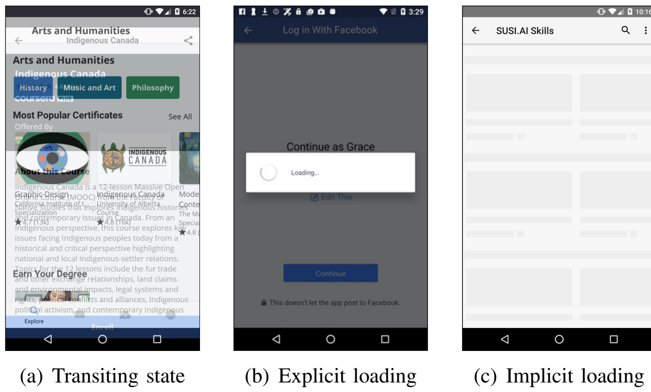
A. Experiment Setup

We collected 32 Android apps as our experimental dataset, which were used in previous studies [22], [23], [24]. They are all top-rated on Google Play, covering different app categories such as news, tools, medical, etc. Details of these apps are shown in our online appendix. These apps do not require logging in, given that logins can be flaky which may affect the experimental measurement. Each app was run for 3 minutes without interruption. Note that we captured a GUI screenshot before triggering each event to visualize what GUI the event executed on.

B. Categorizing GUI rendering state

To understand the GUI states in testing, we conducted a small pilot study on GUI screenshots collected by the commonly-used automated GUI testing tool Droidbot [25], executing with throttle 200ms interval, which is a common setting in automated testing [26]. During the manual examination process, we noticed that there are different types of GUI rendering states, a categorization of these states would help clarify the issues in tools. We recruited two students as annotators by the university's internal slack channel and they

[1] https://github.com/sidongfeng/AdaT



(a) Transiting state (b) Explicit loading (c) Implicit loading

Fig. 2: Examples of partially rendered state.

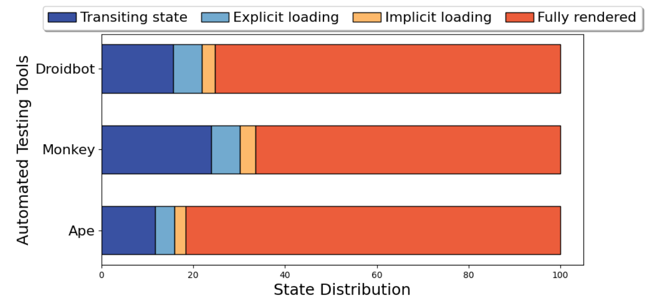


Fig. 3: Distribution of rendering states captured by Droidbot, Monkey, and Ape.

were compensated with $12 per hour. According to the pre-study background survey, they have labeled one UI/UX-related dataset (e.g., GUI element bounding box). To ensure accurate annotations, the process started with initial training. First, we asked them to read a document [13] that outlines the GUI rendering process. Second, we provided an example set of annotated GUIs where the rendering states have been labeled by authors. This enforces a deeper understanding of the GUI rendering states. Third, we asked them to pass an assessment test, which includes a set of test GUIs. Finally, we asked them to manually check 1,500 random GUIs and classified them into four categories following the Card Sorting [27] method:

Fully Rendered State. A fully rendered state represents a GUI rendered completely with all resources loaded and displayed.

Transiting State. As shown in Fig. 2(a), one state is transiting to the next state. As the transition between states takes longer than the throttle interval, two GUIs are overlapped with each other. There are mainly two reasons for capturing transiting state. First, the throttle setting is too short to get GUI fully rendered. Second, there may be issues with the app development (e.g., too many animations, defects in the hardware acceleration), resulting in an unexpected long rendering process.

Explicit Loading State. As shown in Fig. 2(b), it shows an explicit loading state, depicting a loading bar in the GUI, such as a spinning wheel, linear progressing bar, textual hint, etc. It explicitly indicates the process or rendering is in progress and is often used for secure data transformations, such as logging accounts, transferring money, uploading a file, etc. During the explicitly loading state, the GUI is non-interactive.

Implicit Loading State. As shown in Fig. 2(c), some resources are not showing due to network latency or resource defects. Note that, for the explicit loading state, there is always a loading bar; while for the implicit loading state, the loading resources need to be recognized accordingly. For example, in Fig. 2(c), the loading resources appear as some gray layouts.

Summary: By conducting a pilot study on GUIs collected by Droidbot, we categorize four types of GUI rendering states that lie into fully rendered states, and partially rendered states (e.g., transiting state, explicit loading state, and implicit loading state)

C. Are partially rendered states common in testing tools?

To investigate whether testing on partially rendered states is ubiquitous in existing tools, we audited the GUI screenshots captured from three commonly-used testing tools, including Droidbot [25], Monkey [1], and Ape [5], with 200ms throttle interval. In total, we obtained 875, 2,830, and 1,646 GUI screenshots from Droidbot, Monkey, and Ape, respectively. According to our GUI categories observed in Section II-B, the two annotators first annotated the GUI screenshots independently without any discussion, and then met and discussed the discrepancies until consensus was reached.

Fig. 3 depicts the results, showing percentages of GUI state categories in testing tools. We can see that all of the testing tools have the issues of testing events on partially rendered states, i.e., 23%, 32%, 17% exist in Droidbot, Monkey, and Ape, respectively. These partially rendered states will potentially reduce the testing effectiveness. As we can see that state transition is a major partially rendered problem arises in automated testing tools, e.g., 15%, 23%, 11%, in Droidbot, Monkey, and Ape. This indicates that some GUI transitions need more time (longer than 200ms) to finish the transition. The consecutive actions by the testing tool to the incomplete rendered GUI may not trigger the expected event, resulting in a decrease of testing coverage.

Summary: By analyzing three commonly-used testing tools, we find that they all encounter the issue with partially rendered states, which may negatively influence the effectiveness when testing.

D. How to avoid partially rendered states?

To address the issue of partially rendered GUIs, the simplest way is to set a longer throttle interval, extending the inter-event time for transiting or loading. Therefore, we investigated how throttle affects the testing tool by running Droidbot with 5 different throttle intervals, including 200ms, 400ms, 600ms, 800ms, and 1000ms. We further annotated the GUI screenshots following the procedure in Section II-C.

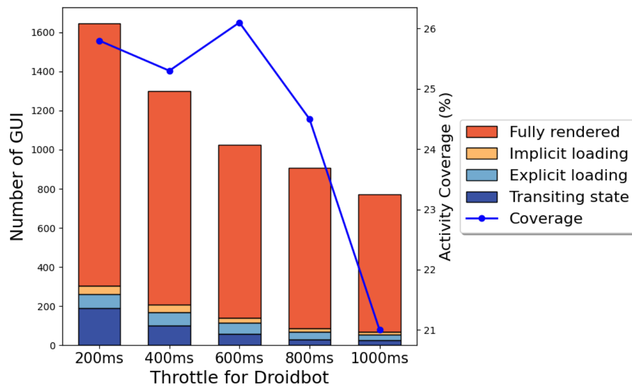


Fig. 4: Number of GUIs and activity coverage in different throttle settings of Droidbot.

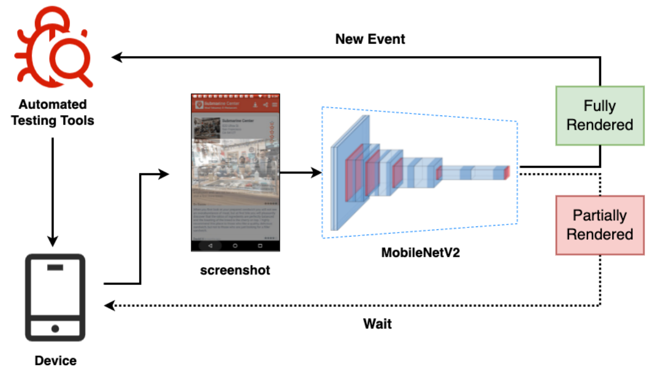


Fig. 5: Overview of our approach.

Fig. 4 depicts the results, showing the number of GUIs and the activity coverage. We can observe that by extending the throttle intervals, the issue with partially rendered states is mitigated, i.e., 17%, 15%, 14%, 9%, 8% incomplete rendering for the throttles from 200ms to 1000ms, respectively. Specifically, the issue of transiting states substantially decreases. This indicates the GUIs can ease of transiting and loading between events with longer intervals. However, utilizing longer throttle intervals decreases the number of GUIs, e.g., 1,646, 1,299, 1,023, 907, 776 GUIs for throttles from 200ms to 1000ms, indicating fewer events executed at the run-time. Consequently, the activity coverage constantly drops, except for the 600ms throttle. This is because when the testing is over-stressed (e.g., 200ms, 400ms), the tool may encounter the issues of the partially rendered state, leading to ineffective testing; when the test throttling is appropriate (e.g., 600ms), the tool sends events to fully rendered GUIs, exploring more activities; when the testing is less-stressed (e.g., 800ms, 1000ms), the tool may stagnate testing, leading to inefficient testing. Therefore, a suitable throttle should strike a balance between effectiveness and efficiency.

Summary: By analyzing five different throttle intervals, we find that extending throttle can help address the issue with partially rendered states. However, an excessive long throttle can reduce the efficiency of automated exploration.

E. Why makes throttle adaptive?

These findings confirm the importance of throttle setting to automated testing, and motivate us to design an approach for balancing effectiveness and efficiency. While the app under testing is mostly idle, the tool has to wait until the GUI finishes rendering before moving to the next event. Taken in this sense, it is worthwhile developing a new effective and efficient method to dynamically adjust the throttle during testing. The underlying issue is to infer GUI rendering states, discriminating partially rendered GUI and fully rendered GUI. Inspired by the fact that these GUIs can be easily classified by human eyes, we propose to identify the GUI rendering states with visual cognitive techniques. As the GUI screenshots are easy to capture for all automated testing tools, our image-based approach is more general and easier to deploy.

III. Approach

This paper proposes a simple but effective approach AdaT to adaptively adjust the throttle base on GUI screenshots. Given that automated testing tools test on the device, we synchronously stream the GUI screenshot capturing, and detect its current rendering state. Based on the GUI rendering inference, we schedule the testing events, which will be sent if the GUI is fully rendered, otherwise, wait explicitly for rendering. The overview of our AdaT is shown in Fig. 5.

The fundamental of AdaT is to adopt a lightweight CNN-based model to classify the GUI rendering state, which is divided into three main phases: (i) the *Data Preparation* phase, which automatically collects a large-scale dataset of partially rendered GUIs and fully rendered GUIs, (ii) the *GUI Rendering State Classification* phase that proposes a CNN-based model to discriminate the current GUI rendering state, and (iii) the *Model Deployment* phase that proposes an efficient deployment of our model in testing tools.

A. Data Preparation

The foundation of understanding GUI rendering state and training deep learning model is big data, whereas manual labeling is prohibitively expensive. The goal of this phase is to automatically collect partially rendered GUIs and fully rendered GUIs by leveraging GUI transiting screencasts, as shown in Fig. 6.

1) GUI Transiting Screencasts: We use the open-sourced Rico dataset [28], which contains 44,418 transiting screencasts from more than 9.7k different Android applications in 27 different app categories. The duration of screencasts spans from 0.5 to 50 seconds. Each screencast contains one or multiple user actions (e.g., tap, scroll) and no-action periods. We discard the transiting periods of scroll action in our dataset. This is because the GUI state can be ambiguous on the development mechanism. For example, scrolling on a lazy-loading GUI may collect partially rendered GUIs; scrolling a pre-loaded GUI may collect fully rendered GUIs. Finally, we obtain 36,038 transiting screencasts.

2) Transiting Frame Identification: A GUI transition is comprised of frames of partially rendered and fully rendered. To identify the frame state in the transiting screencast, we adopt an image processing technique to build a perceptual similarity score for consecutive frame comparison based on Y-Difference (or Y-Diff). YUV is a color space usually used in video encoding, enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using a RGB-representation [29], [30]. Y-Diff is the difference in Y (luminance) values of two images in the YUV color space, used as a major input for the human perception of motion [31].

Consider a transiting screencast $\{f_0, f_1, .., f_{N-1}, f_N\}$, where f_N is the current frame and f_{N-1} is the previous frame. To calculate the Y-Diff of the current frame f_N with the previous f_{N-1}, we first obtain the luminance mask Y_{N-1}, Y_N by splitting the YUV color space converted by the RGB color space. Then, we apply the perceptual comparison metric, Structural Similarity Index (SSIM) [32], to produce a per-pixel similarity value related to the local difference in the average value, the variance, and the correlation of luminances. A SSIM score is a number between 0 and 1, and a higher value indicates a strong level of similarity.

To identify whether one frame is fully or partially rendered, we look into the similarity scores of consecutive frames in the transiting screencast as shown in Fig. 6. The first step is to group frames belonging to the same atomic state according to a tailored pattern analysis. This procedure is necessary because discrete states performed on the screen will persist across several frames, and thus, need to be grouped and segmented accordingly. We find that a fully rendered GUI is in a steady state where the consecutive frames are the same or very similar for a relatively long duration, for example, Fig. 6 (A) and (C). In contrast, a partially rendered GUI shows a great difference on the consecutive frames, revealing an instantaneous transition from one screen to another. For example, as shown in Fig. 6 (B), when the user clicks a button, the current GUI starts to fade out, in which the similarity score starts to drop drastically. Afterwards, the next GUI starts to fade in and the similarity score rises. According to our observation, one common case in partially rendered GUI is that the similarity score becomes steady for a small period of time between two drastically droppings as shown in Fig. 6 (B). The occurrence of this short steady duration is because of the resource loading in GUI, aligning with our observation of implicit loading state in Section II-B. We have empirically set 0.99[2] as the threshold to decide whether two frames are similar, and 5 frames as the threshold to indicate a steady state, in order to differentiate fully rendered and partially rendered GUIs.

3) GUI State Sampling: For each GUI state group, we observe that although the frames are densely recorded in the screencasts, the rendering changes relatively slowly. To prevent bias on redundant data, we propose an approach to sample the GUI frames from GUI groups. As the GUI frames in the fully rendered group are similar or identical, we randomly select one frame. To sample frames from partially rendered groups, we adopt a paradigm using uniform sampling [33] to ensure the diversity in partially rendered GUIs and to prevent the bias of imbalance sampling between fully rendered and partially rendered GUIs. After automated identifying and sampling, we obtain a dataset with 66,233 fully rendered GUIs and 45,623 partially rendered GUIs.

[2]We set up that value by a small-scale pilot study

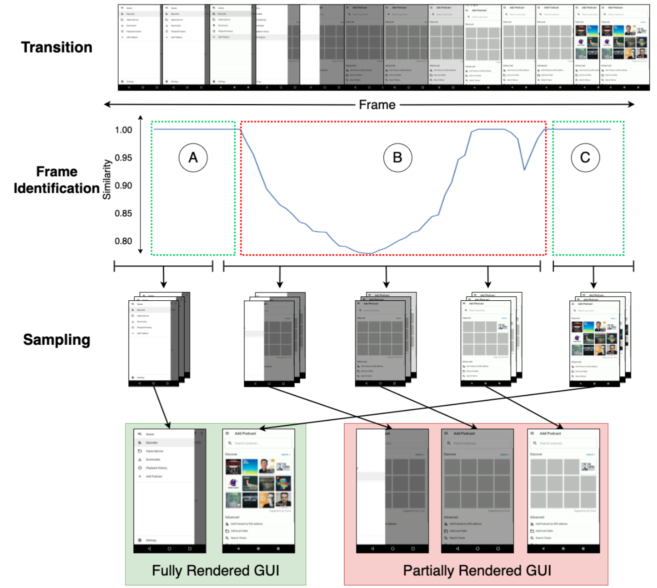


Fig. 6: Pipeline for automated data collection.

B. GUI Rendering State Classification

In this phase, we identify whether the GUI is fully rendered which allows testing tools to execute the next event. To differentiate between fully rendered and partially rendered GUIs, we adopt an implementation of MobileNetV2 [34], which distills the best practices in convolutional network design into a simple architecture that can serve as competitive performance but keep low parameters and mathematical operations to reduce computational cost and memory overhead. In addition to the simulator, the model can even be deployed on mobile devices for efficient testing. This advanced network design speeds up image classification, which is the ultimate goal of this work to efficiently discriminate the GUI rendering states.

Specifically, we adopt a more advanced depthwise separable convolution, combining one $3*3$ convolution layer and two $1*1$ convolution layers to capture essential information from images. We first use a $1*1$ pointwise convolution layer to expand the number of channels in the input feature map. Then, we use a $3*3$ depthwise convolution layer to filter the input feature map and a $1*1$ convolution layer to reduce the number of channels of the feature map. In order to improve the performance and stability between layers, we borrow the idea of residual connection in ResNet [35] to help with the flow of gradients. After the convolution layer, we

add Batch Normalization (BN) [36] to standardize the feature map. Finally, the activation function, Rectified Linear Unit (ReLU), is added to increase the nonlinear properties of the classifier function and of the overall network without affecting the features.

For detailed implementation, we adopt the stride of 2 in the depthwise convolution layer to downsample the feature map. We use ReLU6 defined as $y = min(max(0, x), 6)$, for the first two activation layers because of its robustness in low-precision computation [37]. A linear transformation (also known as Linear Bottleneck Layer) [34] is applied to the last activation layer to prevent ReLU from destroying features. The momentum in the BN layer is set as 0.1. To make our training more stable, we adopt Adam as optimizer [38], and binary CrossEntropyLoss as the loss function [39]. Moreover, to optimize the training model, we apply an adaptive learning scheduler, with an initial rate of 0.01 and decay to half after 10 iterations. For data preprocessing, we resize the GUI screenshots to $768 * 448$. We implement our model based on the PyTorch framework [40]. Note that the hyper-parameter settings are determined empirically by a small-scale experiment.

C. Model Deployment

To make the model efficiently provide feedback on the GUI rendering state to the automated testing tool, synchronization of the GUI and the testing tool is needed. However, capturing and transmitting GUI screenshots can be time-consuming. Therefore, we develop a socket-based smartphone test farm using OpenSTF [41] to stream the real-time GUI screenshot. It is a framework to facilitate the mobile testing process by accessing mobile devices remotely. In detail, the framework consists of three components: *Device Side*, *Server Side*, and *Client Side*. Each component leverages fast and safe microservices to communicate with each other, such as ZeroMQ [42] and Protocol Buffer [43]. The overview of the model deployment is shown in Fig. 7.

The goal of *Device Side* is to monitor and send events to the device as a background process. We utilize the mature and efficient binary method Minicap [44], to capture screenshots on the device. In detail, the screenshots are stored as a binary format, where the first 4 bits represent the screenshot size n, and the next n bits represent the screenshot buffer, for accelerating data transfer between *Device Side* and *Server Side*. The *Server Side* is to keep a device tracker (e.g., daemon) to manage whenever a device is connected or if the device gets disconnected. The *Client Side* leverages the WebSocket to keep receiving the screenshot buffer from the server.

Once the screenshot buffer is received, we decode it into a PyTorch tensor [40]. This tensor is then fed into our trained GUI state classification model to infer the rendering state of the current GUI. If it is fully rendered, we continue to test on the new event, otherwise, we explicitly wait for the next screenshot buffer. To prevent excessive time budget due to the long duration of resource loading or wrong prediction of our model, we set up a maximum waiting threshold. The waiting threshold is empirically set as 1000ms by a small pilot study. We make the model and the source code used to set up AdaT publicly available[3].

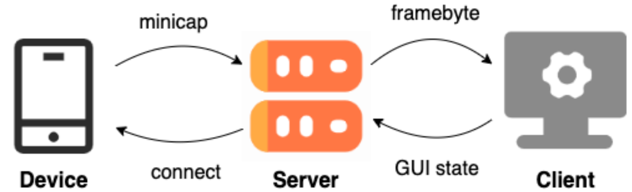


Fig. 7: Overview of model deployment.

IV. Evaluation

The main quality of our study is the extent to whether our AdaT can effectively and efficiently accelerate the automated testing process. To achieve our study goals, we formulate the following three research questions:

- **RQ1**: How accurate and efficient is our model in classifying GUI rendering state?
- **RQ2**: How effective and efficient is our approach in triggering bugs?
- **RQ3**: How useful is our approach when integrated in real-world automated testing tools?

For **RQ1**, we present some general performance of our model for GUI rendering inference and the comparison with state-of-the-art baselines. For **RQ2**, we carry out experiments to check if our tool can speed up the automated GUI testing, without sacrificing the effectiveness of bug triggering. For **RQ3**, we integrate AdaT with DroidBot as an enhanced automated testing tool to measure the ability of our approach in real-world testing environments.

A. RQ1: Performance of Model

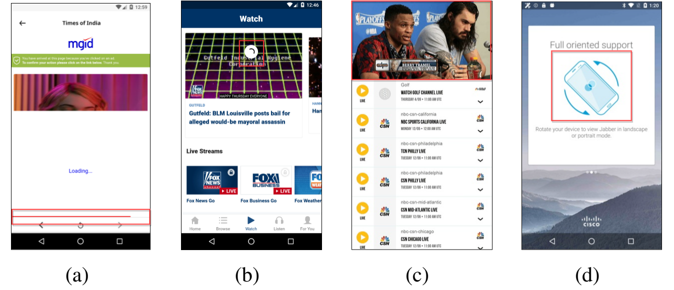
Experimental Setup. To answer RQ1, we first evaluated the ability of our model MobileNetV2 (in Sec. III-B) to accurately and efficiently differentiate between fully rendered GUIs and partially rendered GUIs. To accomplish the evaluation, we followed the procedure to generate the dataset outlined in Section III-A. Regarding our training-testing data split, a simple random split cannot evaluate the model generalizability, as the GUIs in the same app may have very similar visual appearances. To avoid this data leakage problem [45], we split the screens in the dataset by app, completing with the 8:1:1 app split for the training, validation, and testing sets, respectively. The resulting split has 79k GUIs in the training dataset, 10k GUIs in the validation dataset, and 10k GUIs in the testing dataset. The model was trained in an NVIDIA GeForce RTX 2080Ti GPU (16G memory) with 20 epochs for about 3 hours.

Metrics. Since we formulated our problem as an image classification task, we adopted three widely-used metrics i.e., precision, recall, F1-score, to evaluate the accuracy of the model inference. Precision is the proportion of GUIs that are

[3]https://github.com/sidongfeng/AdaT

TABLE I: Performance comparison with baselines

Methods	Precision	Recall	F1-score	Time (ms)
SIFT+SVM	0.763	0.755	0.758	15.81
SIFT+KNN	0.624	0.645	0.634	1.94
SIFT+RF	0.676	0.663	0.669	1.71
SURF+SVM	0.711	0.723	0.716	16.94
SURF+KNN	0.601	0.666	0.631	1.27
SURF+RF	0.650	0.675	0.662	1.29
ORB+SVM	0.674	0.736	0.703	18.10
ORB+KNN	0.601	0.642	0.620	1.15
ORB+RF	0.635	0.657	0.645	1.46
CNN	0.863	0.816	0.838	38.10
AdaT	**0.999**	**0.996**	**0.998**	**43.02**



(a) (b) (c) (d)

Fig. 8: Examples of bad cases in GUI state prediction.

correctly predicted as fully rendered among all GUIs predicted as fully rendered.

$$precision = \frac{\#GUIs\ correctly\ predicted\ as\ fully\ rendered}{\#All\ GUIs\ predicted\ as\ fully\ rendered}$$

Recall is the proportion of GUIs that are correctly predicted as fully rendered among all fully rendered GUIs.

$$recall = \frac{\#GUIs\ correctly\ predicted\ as\ fully\ rendered}{\#All\ fully\ rendered\ GUIs}$$

F1-score (F-score or F-measure) is the harmonic mean of precision and recall, which combine both of the two metrics above.

$$F1-score = \frac{2 \times precision \times recall}{precision + recall}$$

For all metrics, a higher value represents better performance. Since the ultimate goal is to speed up testing process, we also measured the time for inference. For the inference time, a lower time cost represents faster inference of the GUI rendering state.

Baselines. We set up 10 baseline methods, including machine learning-based and deep learning-based, that are widely used in image classification tasks as the baselines to compare with our model. The machine learning-based methods first extract visual features from the GUI screenshots, and then employ a machine learner for the classification. The deep learning-based methods use a convolutional neural network to extract the visual features and then utilize fully connected perceptrons for classification.

In detail, we adopted three types of feature extraction methods used in machine learning, e.g., Scale invariant feature transform (SIFT) [46], Speed up robot features (SURF) [47], and Oriented fast and rotated brief (ORB) [48]. With these features, we applied three commonly-used machine learning classifiers, e.g., Support Vector Machine (SVM) [49], K-Nearest Neighbor (KNN) [50], and Random Forests (RF) [51], for classifying the GUI rendering state. The combination of three types of image features and three classification learning algorithms generated a total of 9 baselines. We also experimented with off-the-shelf feature extraction methods used in deep learning, e.g., traditional CNN with 3 convolutional layers [52]. We set the number of neurons in fully connected layers to 2, representing whether the GUI is in a fully-rendered or partially-rendered state. We trained the baselines following the same procedure of our approach.

Results. Table I depicts the performance of our model MobileNetV2 in classifying the fully rendered GUIs and partially rendered GUIs. The performance of our model is much better than that of other baselines, i.e., 30.9%, 31.9%, 31.6% boost in recall, precision, and F1-score compared with the best machine learning baseline (SIFT+SVM). We observe that the methods based on deep learning perform much better than machine learning due to the reason that the machine learning lacks of feature introspection, as the feature of GUI rendering state varies. Compared with the deep learning baseline, our model further improves 13.6%, 18%, 16% in recall, precision, and F1-score, respectively. In addition, our model takes on average 43.02ms per GUI inference, representing the ability of our model to accurately and efficiently discriminate the GUI rendering state.

Albeit the good performance of our model, we still make wrong predictions for some GUI screenshots. We manually check those wrong GUI cases and summarise two common causes. First, within some GUIs, the representative features are too tiny and inconspicuous to be recognized even with human eyes, for example, the red linear progress bar in Fig. 8(a), and the tiny circular progress bar embedded in the image in Fig. 8(b). Second, some negative data are not really negative data due to the different cognition of GUI state. For example, since Fig 8(c) and 8(d) are screenshots that contain dynamic assets such as videos and gifs, they are automatically annotated as partially rendered GUIs in temporal, while they seem to be fully rendered in static.

B. RQ2: Performance of AdaT

Although we demonstrated the performance of our model in discriminating the rendering state of a single given GUI in the last RQ, it is still unclear if our approach can work in real-world GUI testing. Therefore, we used the existing developer-verified bugs to evaluate the ability of the AdaT to help efficiently test the app without affecting the bug triggering capability.

Experimental Setup. To answer RQ2, we collected 18 crash bugs from 12 Android apps with defects studied in previous works [53]. Each crash bug has a trace script to reproduce, which will constantly trigger the app crash. We evaluated our experiments under the common frame rate 30 fps.

TABLE II: Performance comparison for our tool. "T" denotes the time to trigger the crash in seconds. "R" denotes crash reproducibility.

Crash Bug	Step	Throttle 200ms		Throttle 400ms		Throttle 600ms		Throttle 800ms		Throttle 1000ms		Themis [53]		Consecutive Frame		Asynchronous Streaming		AdaT	
		T	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R
ActivityDiary#118	10	12.77	✓	15.45	✓	17.31	✓	18.68	✓	20.54	✓	26.80	✓	21.74	✓	17.56	✓	16.68	✓
ActivityDiary#285	10	13.99	✗	15.79	✓	17.22	✓	18.97	✓	21.31	✓	34.49	✓	24.11	✓	18.66	✓	15.80	✓
AmazeFileManager#1796	12	14.99	✓	17.03	✓	19.25	✓	21.49	✓	23.89	✓	51.46	✓	35.42	✓	18.28	✓	17.63	✓
AmazeFileManager#1837	4	3.70	✓	4.21	✓	4.87	✓	5.40	✓	6.56	✓	10.41	✓	15.62	✓	6.08	✓	5.48	✓
and-bible#261	18	20.31	✗	23.63	✗	27.29	✗	30.50	✗	33.88	✗	84.24	✓	46.45	✗	39.12	✓	36.01	✓
AnkiDroid#4200	13	14.54	✗	16.90	✓	19.94	✓	22.29	✓	24.04	✓	28.91	✓	21.77	✓	17.83	✓	16.86	✓
AnkiDroid#4451	19	23.42	✓	26.77	✓	30.51	✓	34.08	✓	37.64	✓	39.52	✓	42.91	✓	31.88	✓	29.61	✓
AnkiDroid#5638	4	3.58	✓	4.26	✓	4.85	✓	5.41	✓	6.01	✓	12.20	✓	6.33	✓	4.97	✓	4.04	✓
AnkiDroid#5756	16	17.92	✗	21.04	✗	23.67	✗	26.67	✓	29.76	✓	34.91	✓	23.37	✓	21.10	✓	20.73	✓
AnkiDroid#6145	24	38.20	✓	43.17	✓	48.08	✓	52.21	✓	56.94	✓	69.15	✓	46.01	✓	46.05	✓	39.77	✓
APhotoManager#116	3	2.39	✗	2.86	✗	3.22	✗	3.63	✗	4.07	✓	10.06	✓	4.44	✓	2.88	✓	2.38	✓
collect#3222	9	9.49	✗	11.26	✗	12.74	✓	14.40	✓	16.00	✓	18.34	✓	13.14	✓	11.85	✓	10.59	✓
geohashdroid#118	4	3.67	✗	4.18	✗	5.11	✗	5.40	✗	6.06	✗	12.49	✓	13.33	✓	10.72	✓	9.89	✓
Omni-Notes#745	11	13.82	✓	15.79	✓	18.37	✓	19.57	✓	21.88	✓	28.94	✓	23.42	✓	18.60	✓	17.36	✓
open-event-attendee#2198	5	5.19	✗	5.49	✗	6.30	✓	7.27	✓	8.09	✓	14.66	✓	8.12	✓	6.59	✓	6.33	✓
openlauncher#67	4	5.21	✗	5.80	✓	6.38	✓	7.01	✓	7.61	✓	8.95	✓	6.68	✓	5.76	✓	5.31	✓
Scarlet-Notes#114	23	27.92	✗	32.50	✓	37.06	✓	40.80	✓	45.87	✓	67.78	✓	54.13	✓	34.61	✓	30.10	✓
WordPress#10302	3	2.48	✓	2.87	✓	3.29	✓	3.67	✓	4.09	✓	10.25	✓	5.10	✓	2.77	✓	2.32	✓
Average	10.6	12.97	44%	14.94	66%	16.97	77%	18.75	83%	20.79	89%	31.31	100%	22.89	94%	17.51	100%	15.93	100%

Metrics. To measure the performance of our approach, we employed two evaluation metrics, i.e., whether the method can successfully reproduce the crash bug (**R**), and the time it takes to trigger the bug (**T**). The less time it takes, the more efficient the method can trigger the bugs.

Baselines. We set up 6 throttling methods as our baselines to compare with our AdaT. *Throttle@k* is the fixed interval of k milliseconds between events. We set the throttle k to 200ms, 400ms, 600ms, 800ms, 1000ms, as these throttle intervals are empirically used in automated testing [26]. *Themis* [53] is the benchmark method of our experimental testing dataset. It adopts a widely-used Android testing framework UIAutomator [54], which explicitly waits between events until all resources are acquired.

In addition, we also add two derivatives of our approach to demonstrate the impact of each component. In Section III-A, we utilized heuristic image processing-based methods to calculate the similarity of consecutive frames to automatically discriminate the GUI rendering state. Therefore, we set up one baseline called *Consecutive Frame* based on multiple screenshots to compare with our method based on one single GUI screenshot. In addition, to demonstrate the strength of real-time GUI state streaming outlined in Section III-C, we also conducted an ablation study, namely *Asynchronous Streaming*. Specifically, it leverages native ADB built-in function [55], for example, it first adopts `adb screencap` to capture the GUI screenshot to the device, and then `adb pull` to transmit to the local machine for GUI rendering state classification, asynchronously.

Results. Table II shows detailed results of the time and reproducibility rate for each crash bug, where the number of steps to trigger the bug of each app is also displayed. It takes AdaT 15.93 seconds on average to reproduce all the bugs. Instead, it takes the throttle methods of 200ms, 400ms, 600ms, 800ms, and 1000ms time intervals, on average 12.97, 14.94, 16.97, 18.75, and 20.79 seconds to reproduce the bugs, respectively. The former three throttles only trigger fewer bugs

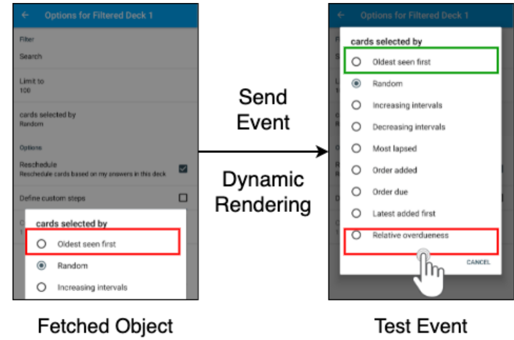


Fig. 9: Example of failure bug detection with short throttle. Red box represents the test location, and green box represents the real object location.

than the latter two, especially the 200ms setting can only trigger 44% of the bugs due to two reasons. First, Fig. 9 shows a failure example to trigger the bugs by using the short throttle (e.g., Throttle 200ms), that the event is executed on an out-of-sync object, due to the GUI is rendering, and the object position is dynamic. Second, short throttle executing the events on partially rendered GUI can throw an unhandled exception bug. However, these bugs will not be encountered by real-world users, instead, they hinder the following app exploration, resulting in the missing of the "real" bugs. In contrast, our approach can efficiently trigger all of the crash bugs by discriminating partially rendered GUIs.

The benchmark method Themis can also trigger all of the bugs, but it takes much longer time, on average 31.31 seconds, which is 2x slower than our approach. This is due to monitoring the potential resources can be time-consuming, including fetching GUI properties (e.g., widget type, location, and size) and explicitly waiting for lazy-loading assets (e.g., video). The more widgets in GUI, the longer it takes to determine the rendering state for Themis, as it monitors the rendering of

all widgets in the GUI. In contrast, we adopt a lightweight approach by leveraging easy-to-obtained GUI screenshots to adaptively adjust the throttle interval between events, obtaining efficiency without affecting the bug triggering capability.

Table II shows the performance of ablation baselines of the AdaT. The heuristic image-processing method (Consecutive Frame) triggers 94% of the crash bugs. One failure case is that the resource loading is so slow beyond our threshold setting in Section III-A, so it requires more frames to determine whether the GUI is fully rendered. In contrast, our approach can trigger all of the crash bugs in a shorter time, i.e., saving 30% time on average. This demonstrates the advantage of our approach of using a single GUI screenshot to discriminate the GUI rendering state, as multiple-screenshots capturing, transmitting, and computing take time. In addition, leveraging a real-time GUI rendering monitor speeds up the testing process (9% faster) than that of an asynchronous monitor (Asynchronous Streaming). As a result, AdaT does not affect the capability to trigger the bugs, especially those caused by partially rendered GUIs; on the other hand, AdaT can speed up the automated testing, saving much of the time budget in hundreds or thousands of steps in long-term testing.

C. RQ3: Usefulness of AdaT

Experimental Setup. To answer RQ3, we carried out a usefulness study to assess the performance of our AdaT within the automated testing tools in the real-world testing environment. To accomplish this, we utilized 32 Android apps which were used in our motivational study in Section II. They are top-rated on Google Play, covering 15 app categories.

Metrics. We employed three evaluation metrics to measure the performance of our AdaT deployed in automated testing tools, i.e., activity coverage and GUIs. For activity coverage, we collected all the activities defined in each app from AndroidManifest.xml following existing studies [56], [57], and measured the percentage of the explored activities for run-time. Note that there may be multiple GUIs with different states in one activity, so we also used the GUIs for evaluation [25]. The number of GUIs represents the number of events sent at run-time, and the number of fully-rendered GUIs represents whether the event is executed on a fully rendered GUI. We also employed number of crashes to evaluate the ability of our tool in bug detection. To ensure the crash validity, we verified them from app developers via issue tracker or direct contact. For all metrics, a higher value represents better performance in automated app testing. We recruited two paid annotators from online posting who have experience in GUI annotation, to annotate the quality of GUIs. To help ensure the validity and consistency of the annotation, we first asked them to spend twenty minutes distinguishing the difference between fully rendered GUIs and partially rendered GUIs. Then, we assigned the set of captured GUI screenshots to them to annotate independently without any discussion. After the annotation, the annotators met and sanity corrected the subtle discrepancies. Any disagreement would be handed over to one author for the final decision. Note that the annotators and the author do not know whether the GUI is captured from our approach or baselines.

TABLE III: Usefulness in real-world testing tool. "FR" denotes the fully rendered GUIs.

Method	Throttle (ms)	Coverage	# Crashes	FR GUI (Total)
Droidbot	200	36.19%	16	2,194 (2,832)
	400	34.47%	15	1,758 (2,199)
	600	35.97%	15	1,257 (1,501)
	800	35.11%	14	750 (859)
	1000	30.15%	9	462 (504)
Droidbot+AdaT	adaptive	**43.14%**	**21**	**2,848 (3,207)**

Baselines. To demonstrate how our AdaT can enhance real-world testing environments, we integrated our approach into the mature automated testing tool Droidbot [25], namely Droidbot+AdaT. In detail, Droidbot+AdaT does not need to carefully set the throttle interval between events. It is adaptive to GUI rendering, moving to the next event as soon as GUI rendering is complete. We have released our integrated version of Droidbot to public[4]. We set up 5 throttle intervals as our baselines, including 200ms, 400ms, 600ms, 800ms, and 1000ms. Each method runs on the app for 10 minutes without interruption. To ensure the validity of the comparison, we used the configurations in Droidbot, such as using greedy depth-first to search activities, randomly generating events, etc.

Results. Table III shows the results of the performance between Droidbot and Droidbot+AdaT. Droidbot+AdaT achieves a median activity coverage of 43.14% across 32 Android apps, which is 6.95% higher even compared with the best baseline (e.g., 36.19% in Throttle 200ms). This is because the dynamic wait in Droidbot+AdaT allows access to more activities that short throttling might disrupt. For example, when the GUI is in rendering progress, a short throttle testing might execute events on partially rendered GUIs to hinder exploration, or send backward events to abandon exploration. In addition, Droidbot+AdaT outperforms the baselines by exploring 3,207 GUI states, and 88.81% are fully rendered. Overall, the results indicate the effectiveness and efficiency of AdaT-enhanced tool in covering most of the activities, GUI states, and fully rendered GUIs, compared with the vanilla tool. As more activities and states are explored, Droidbot+Ours triggers the most crash bugs (21) compared to the baselines.

V. Threats to Validity

In our experiments evaluating our model, threats to internal validity may arise from the leakage of testing dataset. To mitigate this threat, we proposed the training-testing dataset split on apps, representing an unbiased testing set of GUIs. Another potential confounding factor concerns the quality of dataset used to train, test, and evaluate our model performance. The use of automated mechanism to collect our dataset may generate some noise data. To help mitigate the threat, we trained and evaluated our model in a large-scale dataset of 20,125 GUIs, that training a deep learning-based model

[4]https://github.com/sidongfeng/AdaT

with sufficient good data could tolerate a small amount of noise [58], [59].

The main external threats to the validity of our work are the representative of the apps and the Android testing tools selected to evaluate the usefulness of our approach. To mitigate this threat, we selected the 32 top apps from 15 different categories on the Google Play Store. The selected apps vary greatly in their functionalities. To demonstrate the improvements that our approach can have on Android testing tools, we selected Droidbot as it is widely used in previous studies [18], [7].

Another threat to the validity arises from the randomness of the Android testing tools, apps, and emulators in our study and evaluation. Namely, across different runs of the same tool, app, and emulator, the obtained metrics could change. To mitigate this threat, we ran each pair of tools and apps two times, where each run was performed on a newly-created emulator with the same software and hardware configurations throughout all of the experiments. The results were then from the aggregation of the two runs for each pair of tools and apps.

VI. Discussion

Generality for automated testing tools. Results in RQ3 (Section IV-C) have initially demonstrated the usefulness of our approach in real-world practice when integrated into automated testing tools like Droidbot. Our approach is a purely image-based method, relying only on GUI screenshots. As the GUI screenshots are easy to capture in automated testing tools, our approach should play a bigger role in real-world practice.

Generality across apps and platforms. Supporting tests on native and hybrid apps is a critical task in practice [60]. As the GUI screenshots from different types of apps exert almost no difference, our approach can be generalized in testing different types of apps. Another potential interest in automated testing is to support different platforms, e.g., iOS and Web. While we focus on the Android platform for brevity in this study, the tool can be extended to other platforms. We have conducted a small-scale experiment of 50 GUI screenshots from iOS and Web. Results show that our approach can achieve 92% and 86% F1-score in identifying GUI rendering state. We believe that the performance will be further boosted after fine-tuning. We have released all of our model and source code for reproducibility.

Collections of High-quality GUI dataset. A large-scale of GUI collection is the foundation for many downstream deep-learning based GUI related research such as code generation [61], [57], [62], [63], [64], [65], GUI design [66], [67], [68], [69], [70], [71], [72], GUI testing [73], [74], [75], [76], [77], [78], [79], [80], etc. Existing studies [81] have identified the limitations in the mobile GUI dataset, and attempted to denoise the dataset based on GUI widget classname and GUI layout. Our work complement with noise removal study, as results in RQ3 (Section IV-C) have illustrated the benefit of our approach in collecting high-quality (e.g., fully rendered) GUIs, laying a solid foundation for other works in this direction.

VII. Related Work

As our work is to utilize GUI rendering state to tackle the efficiency issue for accelerating automated GUI testing, we introduce related works in two aspects, i.e., automated GUI testing, and efficiency support for testing.

A. Automated GUI Testing

A growing body of tools has been dedicated to assisting in automated app testing. One of the earliest efforts is Monkey [1], Google's official testing tool for Android apps, intended for generating random user events such as clicks, touches, or gestures, as well as a number of system-level events on the GUI. Subsequent efforts have led to test case generation based on randomness strategies [2], [82], [83], or app artefacts (e.g., activity, source code) [3], [4].

Recent tools [84], [25], [85] leverage dynamic and static analysis to reverse engineer a stochastic model from GUI to generate more robust automated testing. Gu et al. [5] present a GUI event-refinement model, that uses GUI runtime information to evolve an initial model to generate precise events. Su et al. [6] propose Stoat that assigns GUI runtime widgets with different probabilities of being selected to achieve effective testing. Moreover, computer vision techniques have been applied to further improve the effectiveness of automated GUI testing. Degott et al. [8] adopt reinforcement learning to identify valid interactions for a GUI element (e.g., a button allows to be clicked but not dragged) to guide testing. Li et al. [7] take a sequence of GUIs captured from manual events to learn a model to predict human-like interactions on the given app. Different from these approaches that focus on sophisticated GUI algorithms for achieving higher test coverage, our approach aims to accelerate automated testing by scheduling the test events with GUI rendering status inference, leading to substantial testing efficiency and effectiveness improvement.

B. Efficiency Support for Testing

There have been many works trying to improve infrastructure support for the purpose of efficient testing. Hu et al. [10] propose AppDoctor that instruments the target apps using invocations of event handlers to quickly find potential sequences of error-triggering GUIs. Song et al. [11] improve the efficiency of AppDoctor by leveraging direct invocations. Wang et al. [12] propose an Android tool Toller that injects into the testing device to efficiently access GUI layout and execute events. Different from those infrastructure support, we aim to speed up automated testing by adaptive throttling, scheduling the testing events for efficiency improvement.

Adaptive throttling is a common practice for efficient testing on the web. Selenium [86] proposes a feature called Explicit Wait to tell the testing driver to wait for an explicit amount of time until the presence of elements. Similar to Selenium, many tools build this feature for mobile testing, such as Appium [87], UIAutomator [54], etc. Specifically, those tools verify the presence of elements by fetching the view hierarchy of the GUI. However, subsequent studies [81] find that the fetched GUI views may be out of sync, leading to tests

on misaligned or invalid objects. Furthermore, those tools only check the validity of the GUI view hierarchy, while not resources, which may restrict the capability of testing exploration. In contrast, we leverage the GUI as a whole with visual information to dynamically adjust the throttle to schedule the events when the GUI is fully rendered, which is analogous to human viewing and interacting.

VIII. Conclusion

Automated app testing is crucial to improve app quality. Despite the numerous automated testing tools, one often overlooked aspect is the throttle between events. A short throttle may reduce the effectiveness of testing, while a long throttle may reduce the efficiency of testing. To strike the balance, we propose AdaT, a lightweight image-based approach to adaptively adjust the throttle based on the GUI rendering inference. Given the real-time streaming on GUI, AdaT adopts a deep learning model to infer the rendering state to adjust events scheduling, sending events when the GUI is fully rendered. The experiments demonstrate the performance and usefulness of our approach in improving the efficiency and effectiveness of automated testing.

In the future, we will keep improving our AdaT for better efficiency in two aspects. First, we can reduce the computational cost of inference by optimizing network architecture. Second, the process of our AdaT can be accelerated by more advanced infrastructure support. We also want to deploy our approach to mobile devices, so that it can be applied to on-device testing. It can be achieved by quantizating our model into a lite-based model.

IX. Acknowledgement

This work is partially supported by the Monash FIT RSP fund.

References

[1] “Ui/application exerciser monkey,” https://developer.android.com/studio/test/other-testing-tools/monkey, 2022.

[2] K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for android applications,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[3] T. Azim and I. Neamtiu, “Targeted and depth-first exploration for systematic testing of android apps,” in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.

[4] W. Yang, M. R. Prasad, and T. Xie, “A grey-box approach for automated gui-model generation of mobile applications,” in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[5] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, “Practical gui testing of android applications via model abstraction and refinement,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[6] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based gui testing of android apps,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[7] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Humanoid: A deep learning-based approach to automated black-box android app testing,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1070–1073.

[8] C. Degott, N. P. Borges Jr, and A. Zeller, “Learning user interface element interactions,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 296–306.

[9] M. E. Joorabchi, A. Mesbah, and P. Kruchten, “Real challenges in mobile app development,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 15–24.

[10] G. Hu, X. Yuan, Y. Tang, and J. Yang, “Efficiently, effectively detecting mobile app bugs with appdoctor,” in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–15.

[11] W. Song, X. Qian, and J. Huang, “Ehbdroid: Beyond gui testing for android applications,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 27–37.

[12] W. Wang, W. Lam, and T. Xie, “An infrastructure approach to improving effectiveness of android ui testing tools,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 165–176.

[13] “Rendering - android developers,” https://developer.android.com/topic/performance/rendering, 2022.

[14] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: Data-driven seed generation for fuzzing,” in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594.

[15] W. Choi, K. Sen, G. Necul, and W. Wang, “Detreduce: minimizing android gui test suites for regression testing,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 445–455.

[16] W. Choi, G. Necula, and K. Sen, “Guided gui testing of android apps with minimal restart and approximate learning,” *Acm Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013.

[17] S. Negara, N. Esfahani, and R. Buse, “Practical android test recording with espresso test recorder,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 193–202.

[18] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, “Owl eyes: Spotting ui display issues via visual understanding,” in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 398–409.

[19] D. Ran, Z. Li, C. Liu, W. Wang, W. Meng, X. Wu, H. Jin, J. Cui, X. Tang, and T. Xie, “Automated visual testing for mobile apps in an industrial setting,” 2022.

[20] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, “Reinforcement learning for android gui testing,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8.

[21] S. Talebipour, Y. Zhao, L. Dojcilović, C. Li, and N. Medvidović, “Ui test migration across mobile platforms,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 756–767.

[22] S. Feng and C. Chen, “Gifdroid: Automated replay of visual bug reports for android apps,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1045–1057.

[23] ——, “Gifdroid: an automated light-weight tool for replaying visual bug reports,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 95–99.

[24] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, “Translating video recordings of mobile app usages into replayable scenarios,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 309–321.

[25] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Droidbot: a lightweight ui-guided test input generator for android,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[26] P. Patel, G. Srinivasan, S. Rahaman, and I. Neamtiu, “On the effectiveness of random testing for android: or how i learned to stop worrying and love the monkey,” in *Proceedings of the 13th International Workshop on Automation of Software Test*, 2018, pp. 34–37.

[27] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[28] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, “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*, 2017, pp. 845–854.

[29] H. Chen, M. Sun, and E. Steinbach, "Compression of bayer-pattern video sequences using adjusted chroma subsampling," *IEEE transactions on circuits and systems for video technology*, vol. 19, no. 12, pp. 1891–1896, 2009.

[30] R. Sudhir and L. D. S. S. Baboo, "An efficient cbir technique with yuv color space and texture features," *Computer Engineering and Intelligent Systems*, vol. 2, no. 6, pp. 78–85, 2011.

[31] M. Livingstone and D. H. Hubel, *Vision and art: The biology of seeing*. Harry N. Abrams New York, 2002, vol. 2.

[32] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[33] F. Shi, E. Petriu, and R. Laganiere, "Sampling strategies for real-time action recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 2595–2602.

[34] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[36] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[37] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[39] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[41] "Smartphone test farm: Stf," https://openstf.io/, 2022.

[42] "Zeromq," https://zeromq.org/, 2022.

[43] "Protocol buffers — google developers," https://developers.google.com/protocol-buffers, 2022.

[44] "Minicap," https://github.com/openstf/minicap, 2022.

[45] S. Kaufman, S. Rosset, C. Perlich, and O. Stitelman, "Leakage in data mining: Formulation, detection, and avoidance," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 6, no. 4, pp. 1–21, 2012.

[46] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[47] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *European conference on computer vision*. Springer, 2006, pp. 404–417.

[48] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*. Ieee, 2011, pp. 2564–2571.

[49] S. B. Kotsiantis, I. Zaharakis, P. Pintelas *et al.*, "Supervised machine learning: A review of classification techniques," *Emerging artificial intelligence applications in computer engineering*, vol. 160, no. 1, pp. 3–24, 2007.

[50] J. M. Keller, M. R. Gray, and J. A. Givens, "A fuzzy k-nearest neighbor algorithm," *IEEE transactions on systems, man, and cybernetics*, no. 4, pp. 580–585, 1985.

[51] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[52] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[53] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 119–130.

[54] "Ui automator," https://developer.android.com/training/testing/other-components/ui-automator, 2022.

[55] "Android debug bridge (adb)," https://developer.android.com/studio/command-line/adb, 2022.

[56] T. Cai, Z. Zhang, and P. Yang, "Fastbot: A multi-agent model-based test generation system beijing bytedance network technology co., ltd." in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 93–96.

[57] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 596–607.

[58] S. Sukhbaatar, J. Bruna, M. Paluri, L. Bourdev, and R. Fergus, "Training convolutional networks with noisy labels," *arXiv preprint arXiv:1406.2080*, 2014.

[59] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[60] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for android hybrid applications," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 250–261.

[61] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "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*, 2018, pp. 665–676.

[62] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "Gui-squatting attack: Automated generation of android phishing apps," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[63] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," *IEEE Transactions on Software Engineering*, vol. 46, no. 2, pp. 196–221, 2018.

[64] S. Feng, S. Ma, J. Yu, C. Chen, T. Zhou, and Y. Zhen, "Auto-icon: An automated code generation tool for icon designs assisting in ui development," in *26th International Conference on Intelligent User Interfaces*, 2021, pp. 59–69.

[65] S. Feng, M. Jiang, T. Zhou, Y. Zhen, and C. Chen, "Auto-icon+: An automated end-to-end code generation tool for icon designs in ui development," *ACM Transactions on Interactive Intelligent Systems*, vol. 12, no. 4, pp. 1–26, 2022.

[66] C. Chen, S. Feng, Z. Xing, L. Liu, S. Zhao, and J. Wang, "Gallery dc: Design search and knowledge discovery through auto-created gui component gallery," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–22, 2019.

[67] J. Chen, C. Chen, Z. Xing, X. Xia, L. Zhu, J. Grundy, and J. Wang, "Wireframe-based ui design search through image autoencoder," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–31, 2020.

[68] C. Chen, S. Feng, Z. Liu, Z. Xing, and S. Zhao, "From lost to found: Discover missing ui design semantics through recovering missing tags," *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, no. CSCW2, pp. 1–22, 2020.

[69] S. P. Reiss, Y. Miao, and Q. Xin, "Seeking the user interface," *Automated Software Engineering*, vol. 25, no. 1, pp. 157–193, 2018.

[70] D. Zhao, Z. Xing, C. Chen, X. Xia, and G. Li, "Actionnet: Vision-based workflow action recognition from programming screencasts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 350–361.

[71] S. Feng, C. Chen, and Z. Xing, "Gallery dc: Auto-created gui component gallery for design search and knowledge discovery," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 80–84.

[72] S. Chen, L. Fan, C. Chen, and Y. Liu, "Automatically distilling storyboard with rich features for android apps," *IEEE Transactions on Software Engineering*, 2022.

[73] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, "Automated reporting of gui design violations for mobile apps," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 165–175.

[74] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, and G. Li, "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*, 2020, pp. 1202–1214.

[75] Y. Su, C. Chen, J. Wang, Z. Liu, D. Wang, S. Li, and Q. Wang, "The metamorphosis: Automatic detection of scaling issues for mobile apps," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[76] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Nighthawk: Fully automated localizing ui display issues via visual understanding," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 403–418, 2022.

[77] D. Zhao, Z. Xing, C. Chen, X. Xu, L. Zhu, G. Li, and J. Wang, "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, 2020, pp. 1286–1297.

[78] B. Yang, Z. Xing, X. Xia, C. Chen, D. Ye, and S. Li, "Don't do that! hunting down visual design smells in complex uis against design guidelines," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 761–772.

[79] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen, "Uied: a hybrid tool for gui element detection," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1655–1659.

[80] M. Xie, Z. Xing, S. Feng, X. Xu, L. Zhu, and C. Chen, "Psychologically-inspired, unsupervised inference of perceptual groups of gui widgets from gui images," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 332–343.

[81] Y. Li, J. He, X. Zhou, Y. Zhang, and J. Baldridge, "Mapping natural language instructions to mobile ui action sequences," *arXiv preprint arXiv:2005.03776*, 2020.

[82] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[83] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013, pp. 68–74.

[84] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Guided bug crush: Assist manual gui testing of android apps via hint moves," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–14.

[85] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 258–261.

[86] "Selenium," https://www.selenium.dev/, 2022.

[87] "Appium," http://appium.io/, 2022.