cvrip: A Visual GUI Ripping Framework*

Heji Huang Ju Qian Wenduo Jia Yiming Jin
Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
jqian@nuaa.edu.cn

ABSTRACT

GUI ripping explores the graphical user interface of an application to build a model which can express the application behavior. The ripped GUI model is useful in various software engineering tasks. Traditional GUI ripping techniques depends on the underlying GUI frameworks to provide the GUI structure information. They are difficult to work across platforms or on non-native applications where the GUI structure information cannot easily be obtained. This work introduces cvrip, a visual GUI ripping framework, to address the problem. cvrip visually analyzes the GUI screen for ripping and does not rely on the underlying GUI frameworks. We introduce many new techniques to enable efficient visual GUI ripping, e.g., a YOLO v5 based model to detect executable widgets, an state recognition acceleration method for fast model updating, and several GUI exploration strategies taking the characteristics of imperfect visual analysis into account. Experiments are conducted to evaluate many technique choices in visual GUI ripping and compare the solution with the traditional style ripping. The results show that cvrip can get exploration coverage competitive to that of the traditional approaches. This suggests visual GUI ripping is a direction worthy of more future studies.

KEYWORDS

model-based testing; GUI testing; GUI ripping; computer vision

1. Introduction

The behavior of a GUI application can be represented by a model consisting of GUI states and state transitions. Many tasks use such models as bases, e.g., test case generation [4], test script repairing [7], fault diagnosis [16], and GUI search [3]. The GUI models can be obtained via dynamic exploration, the process of which is also known as GUI ripping [27]. GUI ripping finds widgets on a GUI state, triggers actions on these widgets to get the next state, and repeats this process to build a GUI model gradually. The existing GUI ripping techniques (e.g., [1][9][17][20][24][32]) require support from the underlying system to obtain internal GUI structure information, such as the set of widgets on a GUI screen and the hierarchy of GUI components. They are hard to work across different systems or GUI frameworks, for example, on both native GUI components and non-native WebView components in Android.

Recently, visual analyses of GUI applications have gained increasing attention. For example, UIED [8] detects GUI widgets on a screen via computer vision. GIFdroid [16] exploits the clues of GUI actions in bug report videos to reproduce bugs. In [36], visual GUI screen analyses are used to optimize random testing of Java Swing applications. The rapid development of computer vision techniques also makes visual GUI ripping possible. Visual GUI ripping extracts GUI states and transitions based on screenshot analysis. It has little dependence on the underlying GUI frameworks and hence can easily be applied on different platforms for whatever native or non-native GUI components.

Although the overall idea of visual GUI ripping is straightforward, to implement such a system, many problems still need to be solved, and many strategies need to be discussed.

1) *How to obtain the widgets on a GUI screen*? There are already many techniques to visually detect GUI widgets from a screen [8]. However, the existing work cannot distinguish executable and non-

executable widgets and only evaluates their effects on single-screenshot analysis. The effectiveness of different methods on the whole executable-widget-focused GUI ripping process is unclear.

2) *How to recognize GUI states*? Visual GUI ripping needs to compare GUI screenshots to identify the current state of an application. Image comparison is often time-consuming. How to compare, and how to reduce the comparison cost in a large screenshot space in order to quickly update the GUI model?

3) *How to explore a GUI application*? Are the traditional GUI exploration heuristics still applicable to visual GUI ripping and have good performance? How to handle the new situations in visual GUI analysis that falsely report or miss-detect GUI widgets? How to prevent cross-app exploration without the GUI implementation information?

4) *How about the performance of visual GUI ripping compared with the traditional GUI ripping methods in terms of the ripping speed and the ripping coverage*?

Keeping such problems in mind, this work presents cvrip, a visual GUI ripping framework which introduces a set of techniques for GUI ripping under computer vision (CV). We also evaluate cvrip on Android applications to experimentally investigate the impacts of different GUI ripping strategies.

In more detail, for GUI widget detection, we introduce a new method on top of the YOLO v5 object detection framework. The method trains models based on an optimized dataset and predicate executable and non-executable widgets on a screen instead of type-distinguished widgets like Buttons and ImageViews. The work then experimentally compares a method based on edge and contour analysis, UIED [8], and the YOLO-based methods under the GUI ripping task. Experiments show that our new method works best in the GUI ripping context.

For GUI state recognition, we experimentally compare methods based on image comparison algorithms SSIM [35] and SSIM+ORB [16] and discuss the impacts of image resizing rates and the image comparison thresholds on GUI ripping. On that base, a state recognition method based on SSIM, resizing screenshots to 1/2 size and using 0.8 as the image compassion threshold, is suggested for use in practice. To speed up the GUI state recognition and GUI model updating, cvrip also introduces an abstract-feature-based state recognition acceleration method. The method can effectively avoid speed degradation of the GUI ripping, especially for applications with rich GUI states.

For the GUI exploration strategies, cvrip follows a click-first exploration strategy because whether actions like swipe, long-press, and drag are valid on a widget is hard to be accurately determined under computer vision. It triggers actions first on the widgets detected with high confidence and then on the low-confidence ones to reduce the negative impacts of false widget detection reports. The tool also adds random position action triggering to reduce the negative impacts of incomplete widget detection. We present a technique based on pre-defined cross-app screens to prevent cross-app exploration. The work brings widget selection strategies like random selection, adaptive random selection, and equivalence grouping into visual GUI ripping. A non-large-text-block first widget selection strategy is also introduced for a new situation in visual GUI ripping. The experimental results show that heuristics like adaptive random selection still work in the visual ripping, and a combined strategy taking many heuristics into account works better than just incorporating a single heuristic.

Compared with the traditional methods, our experiments show that model-based ripping can explore GUI states faster than a random exploration based on visually detected widgets. On the tested subjects, cvrip can achieve more exploration coverage compared with two state-of-the-art tools within the same action steps, although it takes more time to finish these steps.

In conclusion, the main contributions of this work are two folds:

(1) a computer-vision-based GUI ripping framework armed with techniques including YOLO-v5-based executable widget detection, state recognition acceleration, screen-image-based cross-app exploration prevention, and so on.

(2) experimental evaluation of many strategies that affect visual GUI ripping, including the strategies on GUI widget detection, state abstraction, state recognition acceleration, and unclicked widget selection in action firing, which provides a reference for the application of the visual GUI ripping tool.

The rest of this paper is organized as follows. Section 2 presents the overall GUI ripping framework. Section 3 discusses how to detect GUI widgets from screenshots. We present the state recognition methods and the GUI model exploration strategies in Sections 4 and 5. Section 6 is the evaluation. Finally, we show the related work in Section 7 and conclude the paper in Section 8.

2. The Visual GUI Ripping Framework

cvrip explores a GUI application based on its runtime screenshots. It analyzes the screenshots to detect GUI widgets on a screen and identify the current state of the subject application. On each GUI state, cvrip goes forward by triggering GUI actions on the detected widgets and eventually obtains a GUI transition graph (UTG) to model the application behavior (see the top of Fig. 1).

A UTG is a directed graph $G = \langle S, E\rangle$, where S is a set of GUI states and E is a set of state transitions. A GUI state abstracts many concrete screen states of an application. For example, for a screen with a time display, when the time changes from 9:00 to 10:00, the concrete screen state becomes different, but the GUI state is usually considered unchanged because these two screens have equivalent behavior. An edge between two states represents the transition from one GUI state to another. More specifically, an edge $s_i \xrightarrow{w,a} s_j$ in a UTG denotes triggering an action a on a widget w in state s_i can transfer the GUI state to s_j.

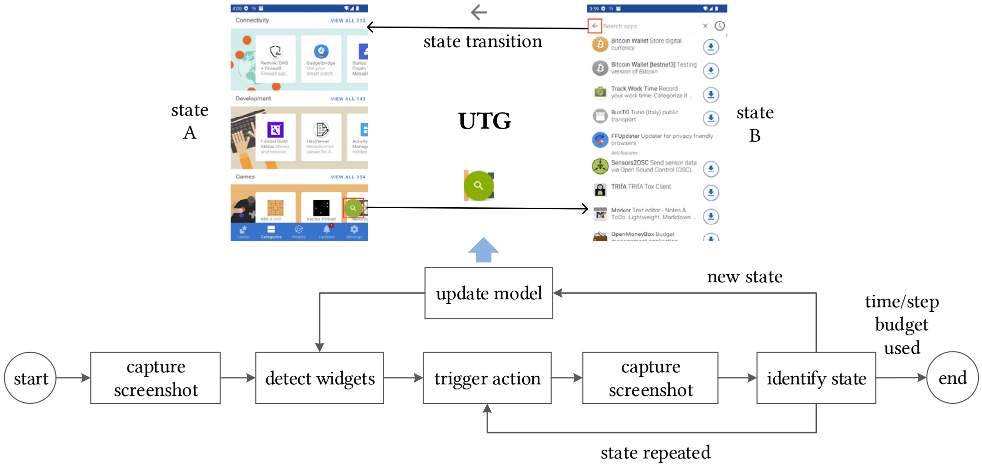


Fig. 1. The workflow of the visual GUI ripping

Fig. 1 presents the workflow of cvrip. The ripping first gets the current screenshot of the subject application and detects GUI widgets on it. It then picks a widget to trigger a GUI action and gets the post-action GUI screenshot. For the new screenshot, cvrip determines whether it belongs to the same GUI state as the last screenshot. If so, the GUI action leads to no state transition. We retry action triggering on other widgets to continue the exploration from the current state or restart the app to continue the exploration from the initial GUI state. If not, we add a new GUI state to the UTG and continue the exploration from this new state. The process repeats until no new state can be found or the budget for exploration steps or exploration time is fully used. After exploration, a UTG will be exported.

3. GUI Widget Detection

GUI widgets can be detected via old fashion CV techniques or deep learning methods. Both work well in some contexts [8]. The widget detection precision and recall affect the model exploration efficiency and comprehensiveness. This work introduces a new widget detection method and experimentally evaluates the performance of different methods in GUI ripping to guide the use of these methods.

3.1. Basic Widget Detection

We first use edge and contour analysis [33], YOLO v3 deep-learning-based object detection [8], and UIED [8] to detect GUI widgets on a screen. Considering that there are many limitations in the existing YOLO v3 widget detection model, we also propose a new widget detection method based on YOLO v5.

3.1.1. Edge and Contour Analysis Based Widget Detection

Edge and contour analysis [33] is a classic old-fashion method to detect objects in an image. Although simple, the method may still get competitive widget detection performance compared with those not well-trained deep learning methods. Therefore, we take it as a candidate choice for GUI widget detection. We refer to the method as *canny* in this paper. *canny* consists of 4 steps: grayscaling based on adaptive thresholding, Canny edge detection, edge enhancement based on Gaussian filtering and dilation operations, and contour extraction. The bounding area of each detected contour is regarded as a potential widget. The method does not distinguish widget types. Each widget is considered to possibly respond to any action like click and long-click on it.

3.1.2. Widget-Type-Distinguished YOLO v3

Previous work [8] trains a YOLO v3 widget detection model (referred to as *yolov*3 for short in this paper) based on the RICO dataset [13]. Each detected widget has a type *t*, a bounding box *b*, a widget type prediction confidence c_t, and a widget area detection confidence c_b. To avoid too many false reports affecting the GUI ripping efficiency, we pick widgets with $c_b \geq 0.8$ as the widget detection results. Since most widget types allow programmers to define event handlers for actions such as click, long-click, and drag on them, and the swipe action can be regarded as an action defined on a screen area rather than on a single widget, we ignore the types of widgets during GUI ripping.

3.1.3. UIED

UIED is a widget detection method introduced in [8] with high detection precision and recall. It outperforms *yolov*3 on the RICO dataset. This work also uses UIED to detect GUI widgets.

3.1.4. YOLO v5 for Executable Widgets

*yolov*3 does not have excellent performance in widget detection [8]. There are at least two potential reasons. First, YOLO v3 has its own technique limitations, especially compared with its later versions. Second, the model was trained on a dataset with many noises. Besides, in GUI testing, only executable widgets that can respond to user actions are of concern, while *yolov*3 cannot distinguish between executable and non-executable widgets. To address these limitations, we also train a YOLO v5 model (*yolov*5 for short) for better widget detection performance.

YOLO v5 integrates more optimization techniques than YOLO v3 for object detection. Our YOLO v5 model simplifies the prediction classes in *yolov*3 and only classifies widgets into two types: executable and non-executable. In Android, usually, widgets with any property in *clickable*, *long-clickable*, *checkable*, and *scrollable* are considered executable [2]. This work does not consider scrollable widgets executable. The reason is that cvrip triggers swipe actions on screen areas instead of a specific widget; hence, it is unnecessary to detect scrollable widgets. Unlike *yolov*3, we do not distinguish fine-grained widget types, e.g., Button and ImageView, because such distinguishment often does not bring differences for GUI action triggering since most widget types allow defining various actions on them.

We follow the existing work [8] to train the YOLO v5 model based on the RICO dataset. RICO has many false reports of widget properties, which lead to invisible widgets in the training ground truth. We first filter out those invisible widgets and then identify the executable widgets from them.

On the GUI hierarchy data provided by RICO, visible widgets are in general distinguished by the *enabled* and *visible-to-user* properties of a widget. This works for most cases. However, when GUI elements are stacked, sometimes an invisible lower-layer element covered by an upper-layer element still has a *visible-to-user* property. From the RICO GUI hierarchy data, we do not know which stacked elements are on the top. This makes it impossible to determine the visibility of these lower-layer elements. In the GUI design tool GUIComp [23], components with types *DrawerLayout*, *SlidingMenu*, and *FanView* are considered with higher displaying priorities and usually cover other components on the view stack. We follow this finding and remove many invisible widgets in RICO. Android applications may have GUI widgets embedded in WebViews. However, such widgets are not included in the GUI hierarchy. To avoid losing widgets in the ground truths, we directly discard the GUI screens containing WebViews. Another problem in RICO is that some view hierarchy data are not synchronized with the screenshots. For such cases, we filter out GUI screens with invalid view hierarchy data following the dataset in [26].

In the identification of executable widgets, sometimes a parent *ViewGroup* component has a *clickable* property, while its child widgets do not have that property, although they can respond to click actions. This is because Android allows a *ViewGroup* to delegate the events on child widgets by onInterceptTouchEvent method. If we just determine the executability according to the widget properties, many executable widgets will be missed. To handle such cases, this work regards a child widget of a ViewGroup component as executable if the ViewGroup is with an *executable* property, but none of its children has that property. Besides the special treatment of ViewGroups, we also heuristically filter out those often non-executable widgets whose resource_ids contain strings like title, header, etc.

We divide the screenshots in RICO by apps into a training set, a validation set, and a test set with 7:2:1 ratio to train the *yolov*5 model. Table 1 compares *yolov*5 with *yolov*3, UIED, and *canny* in the widget detection precision, recall, and F1 scores. The results are obtained on a test dataset containing 1k screenshots from 44 apps (we do not evaluate on RICO to avoid testing on training samples). For *yolov*3 and *yolov*5, we select thresholds 0.8 and 0.1 on the bounding box prediction confidence which can lead to their nearly best performance for evaluation. A widget is considered correctly detected if there is a detected bounding box and a ground truth box with their IoU over 0.5 (IoU>0.5).

The result in Table 1 shows that on single screenshots, *yolov*5 has the best widget detection performance. UIED is known to work well for general widget detection [8]. However, it does not have a high detection precision in our context, although with the best recall. This is because we aim to detect executable widgets, while UIED cannot distinguish executable widgets from those non-executable ones.

Table 1 Performance of different widget detection methods

method	precision	recall	F1	time (s)
***yolov*5**	**0.57**	**0.65**	**0.61**	**0.08**
*yolov*3	0.5	0.39	0.44	0.14
uied	0.27	0.71	0.39	0.58
canny	0.33	0.47	0.39	0.04

3.2. Merging with OCR-Based Widget Detection

Many object detection methods have problems in detecting text widgets. Previous work suggests doing OCR (Optical Character Recognition) with a tool like EAST as a complement to detect text widgets [8]. EAST runs slow. cvrip uses PaddleOCR [40], a faster OCR engine with high detection precision, to detect text widgets on a screen. Each OCR-detected text block is considered a text widget. Widgets detected by OCR may overlap with that detected by other methods. For two widgets C_1 and C_2 detected by different methods, if C_1 is approximately inside C_2, or vice versa, we combine them into a single widget. A widget x is considered approximately inside another widget y if x does not exceed the boundary y for over $p_{exceed} = 20$ pixels. If not inside y, we conservatively treat x and y as two widgets.

4. State Recognition and Model Updating

The existing GUI ripping techniques leverage the internal GUI structure information, such as the implementation classes, the widgets on a screen, and the hierarchy of widgets, provided by the underlying system of the application under test to recognize GUI states. When there is no such information, and only the external screenshot is available, how to recognize GUI states becomes a difficult problem and may affect the efficiency and effectiveness of GUI ripping. This section first analyzes the objective of GUI state recognition and then proposes a visual method for distinguishing GUI states.

During GUI ripping, each time a GUI action is triggered, a new screenshot will be obtained. We need to determine whether the new screenshot belongs to an existing or a new state and thus update the GUI mode. Such state recognition requires a large number of comparisons from the new screenshot to that of the existing states. These comparisons are costly. cvrip introduces an abstract-feature-based acceleration technique to speed up the state recognition process. The technique reduces the required detailed screenshot comparisons by first making a rapid comparison on the abstract features of screenshots. The rapid comparison can filter out impossible-to-match UTG states ahead of detailed state comparisons, thereby improving the efficiency of GUI model building.

4.1. GUI State Recognition

GUI state recognition is mainly used to determine whether the current GUI screen has been transferred to a previously visited state, thus avoiding unnecessary repeated visiting of widgets on the same state. State recognition must conform to the semantics of GUI screens. Significantly different GUI screens should not be recognized as the same state. cvrip recognizes GUI states based on image comparison methods. In the literature, SSIM (structural similarity) is an effective and widely used image comparison method [35]. GIFdroid [16] also combines SSIM and ORB (Oriented FAST and Rotated BRIEF) to compare images and distinguish states. This work adopts SSIM and SSIM+ORB as candidate methods for state recognition. We calculate the image similarity and check whether the similarity is over a pre-defined threshold to determine whether two screenshots belong to the same state.

GUI states should be recognized in appropriate granularity. If the recognition is too fine-grained, slightly different GUI screens may be recognized as distinct states. Many GUI behavior triggered on these distinct states might be actually the same. This will affect the model exploration efficiency. If the recognition granularity is too coarse, unexplored GUI screens may be considered explored, which may lead to incomplete GUI ripping. This work controls the state recognition granularity in two ways. One is setting different image comparison thresholds. The other is resizing the images to different scales. We will experimentally investigate the impacts of various parameter settings on GUI model exploration.

4.2. Abstract-Feature-Based State Recognition Acceleration

Assume that there are n states in the current UTG. Given a new screenshot v, if we directly compare v with the existing n states by SSIM to determine whether v belongs to a historical state, the total comparison cost is the time of n SSIM comparisons. A single SSIM comparison often takes 0.1-0.3s. When n reaches tens of hundreds, the time cost will be extremely high.

cvrip uses feature extraction in image retrieval [15] to accelerate the above state identification process. There are two types of methods to extract abstract features from screenshots.

- Classic algorithms, such as *ahash*, *dhash*, and *phash*. These algorithms calculate a hashcode as the abstract feature of an image. Similar images have similar hashcodes. The similarity between the hashcodes can be assessed via Hamming distance. These classic algorithms run fast, but the calculated abstract features may not be representative enough.

- Deep-learning algorithms, like *Resnet*. They use deep networks to calculate a feature vector from an image. The similarity between the abstract features can be evaluated via cosine distance. The deep-learning algorithms run a bit slower but have demonstrated strong abilities in finding similar images.

This work applies both two types of methods in GUI ripping. We select a small portion of historical states with similar abstract features for detailed state comparison based on SSIM or SSIM+ORB. Considering that in the process of GUI ripping, the scale of historical states is usually not too large, we do not apply the vector indexing technique in image retrieval to speed up the finding of similar abstract features. Only a brute-force search is used to achieve the above goals. The detailed process is shown in Fig. 2. First, after obtaining each GUI screenshot, we extract an abstract feature from it and save the feature for reuse. For a new screenshot S with an abstract feature F, we obtain its k nearest neighbor states by checking the similarity between F and the abstract features of the historical states' main screenshots. Then, for the k neighbor states, SSIM or SSIM+ORB is used for detailed state comparison. If the new screenshot S is not considered similar to any historical state with similarity over a pre-defined threshold, it is regarded as a new state and will be added to the UTG. Otherwise, S will be bonded to an existing state, and the UTG need not to be updated.

The original screenshots are in a high dimension of about 10^6, while abstract features are in a low dimension of about 10^3. The cost of abstract feature comparison is much lower than that of the original screenshot. We use fast abstract feature comparison to reduce the required times of detailed screenshot comparison. In this way, the speed of state recognition for a new screenshot can be accelerated.

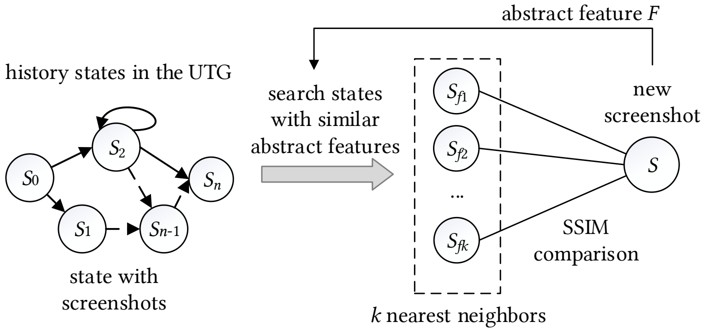


Fig. 2. A demonstration of the state recognition process

5. Visual GUI Model Exploration

GUI ripping depends on certain exploration strategies to determine which action to be performed next based on the current GUI state and exploration history. This paper first presents an overall exploration strategy with new problems like failing to detect GUI widgets, inaccurately detected widget boundaries, and cross-app jumping in visual GUI ripping under concern. Then, for the selection of unvisited widgets on an app screen, we migrate some classic selection strategies to the visual ripping, introduce new strategies, and experimentally evaluate whether these strategies benefit efficient GUI model exploration.

5.1. The Overall Exploration Strategy

For a GUI widget on a touch screen, *click* is the main action, and whether the widget will respond to actions like *swipe* and *long press* is hard to predict from screen images. To avoid invalid GUI actions, especially at the early stage of GUI ripping, we use an overall strategy that first triggers click action on all GUI screens and then triggers swipe, long-press, and other actions to explore the GUI model.

5.1.1. Probability Weight Guided Model Explore

As shown in Fig. 3, there are three possible situations in GUI exploration. One is that not all widgets in a GUI state have been clicked. For this situation, we select one unclicked widget to click. The second is that all the widgets on the current GUI state have been clicked at least once, but there are unclicked widgets on other UTG states. In this case, we select an action to make the exploration directed to the states with more unclicked widgets. The third situation is that all the widgets on the UTG have been clicked once. We trigger actions other than click according to some probability weights for this situation.

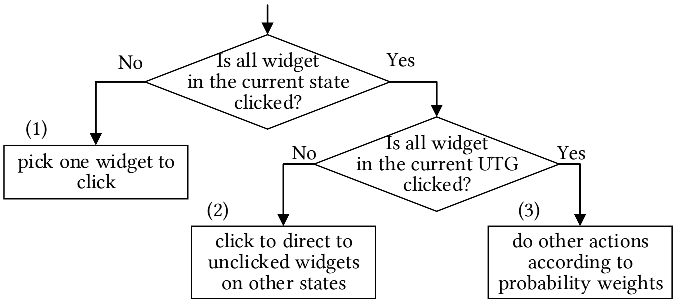

Is all widget in the current state clicked?
No
Yes
(1)
pick one widget to click
Is all widget in the current UTG clicked?
No
Yes
(2)
click to direct to unclicked widgets on other states
(3)
do other actions according to probability weights

Fig. 3. Three situations in GUI model exploration

The first situation will be discussed in more detail in Section 5.2. For the second situation, the candidates for the next actions include clicking on a widget, restarting the app, going back, and randomly clicking on the full screen. We follow Stoat [32] to determine the next action according to some probability weights. The weights of clicking on a widget and restarting the app are calculated by counting how many widgets have not been clicked on the states recursively reached in δ=3 steps of depth-first UTG traversal after doing the action. Let on the current UTG, starting from an action a, the number of unclicked widgets on the state at the first exploration depth (directly reached after the action) is n_1. At the second and third exploration depths, those numbers are n_2 and n_3, respectively. Then, we define the *unexplored content index* of action a be $e_a = n_1 + n_2/2 + n_3/4$. The probability weight of clicking a widget and restarting is set to (w_{base} + e_a/max(e_{max}, 1)), where w_{base}=0.1 and e_{max} is the maximum unexplored content index for all candidate actions. Here we set a non-zero base weight w_{base} because the image-based widget detection is not exactly accurate, and the situation at exploration depth over 3 is unknown. Some unclicked widgets might be missed in the forward look. Even if e_a=0, there still should be some probability assigned to action a to continue the exploration.

If e_{max}= 0, namely all the widgets within δ exploration depth from the current state in the UTG have been clicked once, we set the weights of going back and randomly clicking on the full-screen to

w_{base} * |widgets|, which is equal to the total probability of clicking on widgets. Otherwise, the probability weights of going back and randomly clicking are set to w_{base}, which is the same as clicking on a single widget with all descendant widgets in the δ exploration depth of the UTG being clicked.

For the third situation that all the existing widgets on the UTG have been clicked, the candidate actions include random swiping on the screen, random clicking at any coordinates, going back, random long pressing, and random dragging. We randomly trigger such actions with probability weights 0.4, 0.2, 0.2, 0.1, and 0.1, respectively. The largest probability weight is assigned to swipe since it is widely used in GUI applications.

An action may not necessarily lead to GUI state changes. If the GUI state does not change after γ=5 continuous actions, we consider the exploration may enter a dead end due to bad widget detection, application failures, or else. cvrip will then restart the app to continue exploration from the entry state.

5.1.2. Action Input

The boundary detected for a widget may be inaccurate. For a widget, constantly triggering actions at a fixed position inside it has a risk of never successfully firing the action. Therefore, in GUI exploration, we select a random point in the 70% center region of the widget to trigger actions. For swipe actions, we also generate the start and end points randomly.

5.1.3. Handling False and Missing Reports of Widgets

Image-based widget detection often has false reports. They can lead to invalid actions and affect the efficiency of model exploration. We introduce a result degradation technique to handle this problem. The technique degrades detection results unlikely to be widgets as markers instead of normal widgets. During GUI ripping, we do not visit these markers until all the main widgets have been clicked.

The markers are mainly degraded from the following widgets.

- Widgets with too small widths, heights, or areas, which are unlikely normal widgets;
- Widgets with too small or large width/height aspect ratios, which are often separator bars instead of normal widgets with specific behavior defined on them;
- Widgets detected with low confidence in methods like YOLO v3.

There are also missing reports in image-based widget detection. They may lead to incomprehensive model exploration. To address the problem, on the one hand, after clicking all of the detected widgets in the UTG, we trigger actions at random coordinates according to the strategy introduced in Section 5.1.1 to alleviate the negative effects of missing reports. On the other hand, no widget detected does not mean no widget exists. For a screen with no widget detected, we do not wait until all the widgets in the UTG have been clicked to do random clicking on that screen. Each time reaching such a screen, cvrip triggers GUI action at random coordinates with the following probability weights [back: 0.3, swipe: 0.2, restart: 0.2, click: 0.2, long press: 0.05, drag: 0.05]. This has chances to hit the missed widgets, or we may go back, swipe to another view, or restart the app.

5.1.4. Preventing Cross-App Exploration

Cross-app exploration refers to the situation that the model exploration jumps to another application different from the one under test after a GUI action and continues exploration in that application. Under visual GUI ripping, there is no internal GUI structure information for determining whether the current screen is from another application. We need some other mechanisms to prevent cross-app exploration.

One possible choice is providing the images of widgets which may lead to cross-app jumps before GUI ripping. When triggering GUI actions, we shall exclude the widgets whose images match anyone in the provided image set. However, there are often too many widgets which may cause cross-app jumps. It is hard to list all of them. Besides, matching images on a large image set also takes a long time.

cvrip prevents cross-app exploration in another way. It allows pre-defining a set of full-screen images which may be reached after cross-app jumps, e.g., the Google Play home screen, the system settings screen, etc. The tool then uses the accelerated state comparison method introduced in Section 4.2 to determine whether the current control has been transferred to another application. If true, cvrip restarts the application immediately to return to the original application. Since there are usually not too many targets of cross-app jumps, this method requires only a few cross-app GUI screen images to effectively avoid cross-app exploration.

5.2. Widget Selection on Incompletely Clicked GUI States

Regarding which widget to be clicked next on an incompletely clicked GUI state, there are classic selection strategies such as random selection [10], adaptive random (diversity-first) selection [21], and selecting from equivalence groups [6][32], and intelligent strategies based on techniques like reinforcement learning [29][31]. This preliminary study does not try to find the best selection strategy. Instead, we try to apply the classical strategies and experimentally investigate which heuristics might benefit visual GUI ripping so that more powerful exploration strategies can be invented in the future.

Among the classic strategies, in visual GUI ripping, due to the lack of internal GUI structure information, the equivalence grouping of widgets cannot be conducted in the traditional way. We propose a new method to apply the equivalence grouping strategy. Text widgets bring new challenges for visual GUI ripping. We also introduce a non-large-text-block first strategy to handle such widgets better. Combining heuristics from different perspectives, a strategy taking spacial diversity, equivalence grouping, and text features all into account is proposed as well.

5.2.1. Random Selection Strategy

The random selection strategy randomly picks an unclicked widget for the next GUI action. Although simple, studies show that tools like Money utilizing this strategy still has competitive performance [10].

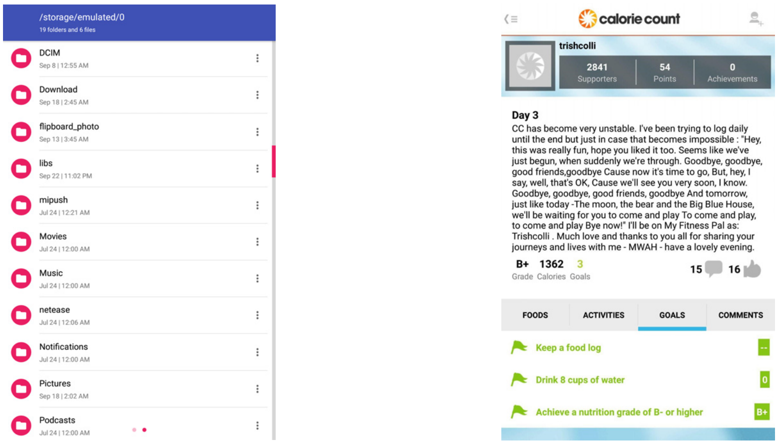
5.2.2. Spatial Diversity First Strategy

Randomly selected widgets may continuously occur in a small screen area and thus cannot quickly expand the exploration scope. Adaptive random testing (ART) [21] can increase the testing diversity and overcome this limitation to some extent. cvrip adopts the idea of ART and supports a spatial diversity first strategy to select widgets. This strategy takes a set W of the unclicked widgets and a set T of the already clicked widgets as the inputs. It then calculates the average Euclidean distance between each widget in W and all the widgets in T. The widget with the farthest distance is selected as the widget to be clicked next. In this way, the clicked widgets can be spatially spread on a GUI screen as quickly as possible, and the GUI behavior might be fastly covered.

5.2.3. Equivalence Grouping Strategy

Widgets in the same List or Grid often have close behavior, e.g., the items in a file list shown in Fig. 4(a). These similar widgets can be classified into equivalence groups. At the early stage of GUI ripping, we can just select one representative widget in each group to trigger GUI actions [6]. After exploring all the groups for once, we may further explore the other widgets in a group. In this way, different GUI behaviors can be explored as fast as possible. We call this the equivalence grouping strategy.

However, when only screenshots are available, the above equivalence groups are difficult to be determined. This work presents a simple method based on alignment, height difference, and height spacing to solve the widget equivalence grouping problem. The method first categorizes all the screen widgets into left-aligned groups. A left-aligned group is then divided into close-height subgroups according to the widget heights, and a close-height group is further divided into equal-spacing groups with neighbor widgets having close vertical spaces between them. The equal-spacing groups are the final equivalence groups. We say widgets are left aligned or with close height, or two vertical spaces are close if the differences between widgets' left boundaries, widget heights, and widget spaces are less than a threshold of d=10px.



/storage/emulated/0
19 folders and 6 files

DCIM
Sep 8 | 12:55 AM

Download
Sep 18 | 2:45 AM

flipboard_photo
Sep 13 | 3:45 AM

libs
Sep 22 | 11:02 PM

mipush
Jul 24 | 12:21 AM

Movies
Jul 24 | 12:00 AM

Music
Jul 24 | 12:00 AM

netease
Jul 24 | 12:06 AM

Notifications
Jul 24 | 12:00 AM

Pictures
Sep 18 | 2:02 AM

Podcasts
Jul 24 | 12:00 AM

(a) list in a file manager

calorie count

trishcolli

2841 Supporters | 54 Points | 0 Achievements

Day 3

CC has become very unstable. I've been trying to log daily until the end but just in case that becomes impossible : "Hey, this was really fun, hope you liked it too. Seems like we've just begun, when suddenly we're through. Goodbye, goodbye, good friends,goodbye Cause now it's time to go, But, hey, I say, well, that's OK, Cause we'll see you very soon, I know. Goodbye, goodbye, good friends, goodbye And tomorrow, just like today -The moon, the bear and the Big Blue House, we'll be waiting for you to come and play To come and play, to come and play Bye now!" I'll be on My Fitness Pal as: Trishcolli . Much love and thanks to you all for sharing your journeys and lives with me - MWAH - have a lovely evening.

B+ Grade 1362 Calories 3 Goals 15 16

FOODS ACTIVITIES GOALS COMMENTS

Keep a food log --

Drink 8 cups of water 0

Achieve a nutrition grade of B- or higher B+

(b) long text paragraph

Fig. 4. Examples for the equivalence grouping and the non-large-text-block first strategies

5.2.4. Non-Large-Text-Block First Strategy

Large text blocks are often for content displaying and unlikely to cause GUI state transitions, e.g., the text paragraph in the middle of Fig. 4 (b). In GUI ripping, it is often unnecessary to trigger meaningless actions like click on these large text blocks. However, in visual analysis, the large text blocks are difficult to be detected even with OCR techniques. This is because OCR often recognizes a long paragraph as multiple broken text pieces. Even if each text line is detected, it is hard to determine whether these lines are as a whole or correspond to individual text widgets such as multiple lines of links.

This paper proposes a non-large-text-block first strategy to handle large text blocks. It regards the text phrases merged from neighbor text pieces in a row with horizontal spaces less than 1/3 of the text height and the text phrases with width/height aspect ratio greater than α (α is set to 18 in this work which corresponds to the length of a relatively long word sequence) as large text blocks. Among all the unclicked widgets, the strategy clicks the non-text widgets and the short texts outside those large text blocks at first. It treats a large text block merged from close text pieces as an equivalence group of the internal text pieces and applies the equivalence grouping strategy on them. In this way, we may avoid too much meaningless text clicking at the beginning of GUI ripping.

5.2.5. Combined Strategy

The combined strategy calculates weights w_{div}, w_{eq}, and w_{text} for each unclicked widget from the spatial diversity, equivalence group, and large text block perspectives, respectively. It then takes $(w_{div}+w_{eq}+w_{text})/3$ as the final probability weight for determining which widget to be clicked next. The spatial diversity weight w_{div} is the ratio between the widget's ART distance and the maximum ART distance of all unclicked widgets. For the equivalence group weight, if a widget is in a never visited equivalence group, its weight is w_{eq} =1. If the widget is in a partially clicked equivalence group, its weight is the product of *sd*, the state derivation index, and *pr*, the proportion of unclicked widgets within the group, i.e., $w_{eq} = sd \times pr$. The state derivation index is n_s, the number of states directly reached from those clicked widgets in the group, divided by n_c, the number of the clicked widgets, i.e., $sd = n_s / n_c$. The larger the index is, the more diversity of the behavior triggered by clicking widgets in the group. For the weight from the large text block perspective, the strategy sets the weight of unclicked non-text widgets and short texts to w_{text} =1. The weight of an unclicked large text block is set to w_{text} =0.6. If a large text block is merged from fragment text pieces and some of the fragments have been clicked, the weight w_{text} of an unclicked text piece is firstly calculated following that of the equivalence groups and then multiplied by 0.6.

6. Evaluation

We implement cvrip based on OpenCV [41]. cvrip currently supports GUI ripping of Android applications with GUI actions triggered via adb. After GUI ripping, a UTG compatible with DroidBot [24] is output and can be viewed in a browser. In the abstract-feature-based state recognition acceleration, we take the resnet50 model trained on ImageNet, widely used in finding similar images, as an option to extract features from screenshots. During model exploration, to effectively obtain the updated screenshots, we wait for 1s after each GUI action and 4s after each application restart.

This work conducts experiments to answer the following research questions.

RQ1: How are the impacts of different widget detection algorithms on the efficiency of GUI ripping?

RQ2: How are the impacts of different state recognition methods on the efficiency of GUI ripping?

RQ3: How is the effectiveness of different abstract-feature-based state recognition acceleration methods in speeding up the GUI model exploration?

RQ4: How are the impacts of different widget selection strategies for unclicked widgets in a GUI state on the efficiency of GUI model exploration?

RQ5: How is the efficiency of cvrip's model-based GUI exploration compared with the existing random computer-vision-based exploration approaches?

RQ6: How is the efficiency of the visual GUI ripping compared with traditional ripping techniques?

6.1. Experimental Setup

The experiment is conducted on 10 Android applications (see Table 1) runnable on virtual machines. These applications cover life, social, news, and other fields. Some of them are selected from previous research. Others are commercial applications or free applications in F-droid with rich GUI interfaces.

Table 2 The experimental Subjects

app	package	description	source
trivago	com.trivago	travel, local	[37]
antennapod	de.danoeh.antennapod	podcast player	[29]
flipboard	flipboard.cn	news	[17]
runnerup	org.runnerup.free	health, fitness	[29]
anymemo	org.liberty.android.fantastischmemo	memo	[17]
amaze	com.amaze.filemanager	file manager	[17]
douban	com.douban.frodo	social media	
markor	net.gsantner.markor	editor, note, todo	[19]
suntimes	com.forrestguice.suntimeswidget	weather	[29]
dartchecker	com.DartChecker	game scoreboard	

We evaluate the efficiency of the GUI ripping by the exploration coverage achieved after 1000 GUI actions. For commercial applications, due to code protection, sometimes it is hard to obtain the source or byte code level coverage. Besides, in GUI ripping, the objective is to explore as many GUI states as possible instead of maximizing the executed code (more explored GUI states may even result in less code coverage). Therefore, this work only follows the multi-level GUI comparison criteria (GUICC) in [2] to obtain coverage at the GUI screen level from five granularities: activity, layout, executable, and content. The activity coverage distinguishes GUI screens from Android activities. The layout coverage distinguishes the covered screens according to the GUI layouts embodied in the non-terminal nodes and their topological relationships in the GUI hierarchy trees. The executable coverage further takes the executable terminal widgets into GUI screen distinguishment on the basis of the layout coverage. The content coverage is the most fine-grained one. For two screens with the same structure, even if only the text contents of some widgets change, they will be viewed as different GUI screens.

Cross-app jumps can not be thoroughly avoided during GUI ripping. The screens reached by such jumps still reflect the exploration scopes under different GUI model exploration strategies in limited action steps. Therefore, when collecting the exploration coverages, we take all the visited GUI screens, including the ones reached by cross-app jumps, into account to evaluate the impacts of different exploration strategies. Meanwhile, the experiment also lists the ratio of the explored inner-app contents to test the ability of our cross-app exploration prevention mechanism.

cvip supports several dimensions of strategies to control the GUI ripping. The space of strategy combination is huge. Doing 1000 steps of GUI ripping for one application under one strategy cost about 1.5 hours. It is almost impossible to test all the strategy combinations. Therefore, when answering a research question, we fix the strategies other than the currently concerned ones into default choices to reduce the experiment cost. For widget detection, this work sets *uied+ocr*, the state-of-the-art widget detection method, as the default choice. For state recognition, a strategy named *ssim*:0.5:0.9 is applied by default. *ssim* denotes using SSIM for image comparison. The number 0.5 in the middle stands for the screenshot resizing rate, and the number 0.9 is the similarity threshold for determining whether two screenshots belong to the same state. The unclicked widgets are, by default, picked via random selection. We use *phash* with a length of about 1k as the default method to abstract features from screenshots for state recognition acceleration.

The coverage achieved for different applications is incomparable. Therefore, we do not directly use the average or total coverage of all the subjects as an indicator to compare different strategies. Instead, this work checks a *superiority index* counting how many apps a strategy can achieve advantages on to compare its effectiveness. Let a strategy get i-th ranked performance among all the compared strategies on r_i subject applications (i∈{1,2,3,4,5}). Then, the superiority index of this strategy is $s = \sum_i k_i r_i$, where $k_i = 2^{1-i}$ is the coefficient at the i-th rank order. Higher rank orders have larger coefficient values.

(1) RQ1: Widget detection

We apply four widget detection methods, *canny+ocr*, *yolov*3*+ocr*, *yolov*5*+ocr*, and *uied+ocr*, and used the achieved coverage to assess different methods' impacts on the GUI exploration efficiency.

(2) RQ2: State recognition

To reduce the cost, the experiment only tests the following state recognition configurations: *ssim*:0.5:0.9, *ssim+orb*:0.5:0.9, *ssim*:0.25:0.9, and *ssim*:0.5:0.8. The first two check the impacts of the image comparison methods SSIM and SSIM+ORB. From the first and the latter two configurations, it is possible to see how different screenshot resizing rates and image comparison thresholds affect GUI ripping. We do not test without screenshot resizing since it is too costly to use the original scale screenshots, and such a high image quality often does not bring significant increments to the exploration coverage. The experiment enables phash-based acceleration while evaluating state recognition methods because, without acceleration, the accumulated GUI ripping time can potentially be very large for some applications.

(3) RQ3: State recognition acceleration

We run configurations with and without state recognition acceleration to evaluate the effectiveness of the abstract-feature-based state recognition acceleration. The tested feature abstraction methods include *phash*, *dhash*, *ahash*, and *resnet* (resnet50). For the three hash methods, a hash length of about 1k fitting the image shape is used.

(4) RQ4: Widget selection

For RQ4, the experiment applies different widget selection strategies introduced in Section 5.2 and checks the achieved coverage to evaluate the impacts of these strategies.

(5) RQ5: Comparision with random CV-based exploration

The work in [36] introduces a random GUI exploration strategy based on widgets detected via computer vision. It randomly fires GUI actions on the detected widgets. However, the technique only supports Java Swing applications. We reimplement its idea under the cvrip framework and take that as a baseline for comparison. The GUI exploration time and coverage under the given action steps are compared to understand the performance of different methods.

(6) RQ6: Comparison with non-CV-based methods

Ape and Fastbot are two state-of-the-art non-CV-based model-based GUI testing tools from academia and industry. We trigger 1000 actions with them and obtain the GUI exploration time and coverage to see if the visual GUI ripping has a competitive performance with the traditional methods.

6.2. Results and Discussion

Table 3 lists the results of cross-app exploration prevention achieved by providing 3-5 excluded cross-app screens. In the table, column *total* presents the average numbers of GUICC screens covered at different abstraction levels under test executions with the default running configuration, and column *inner%* lists the percentage of the covered screens inside the subject application.

Table 3 Results of cross-app exploration prevention

app	activity		layout		executable		content	
	total	inner%	total	inner%	total	inner%	total	inner%
trivago	18	83.3	46	82.6	53	83.0	173	90.2
antennapod	6	50.0	74	60.8	94	62.8	203	66.5
flipboard	24	75.0	237	83.1	264	83.7	425	83.8
runnerup	9	88.9	26	96.2	45	97.8	124	99.2
anymemo	15	86.7	32	93.8	40	95.0	239	99.2
amaze	5	60.0	46	97.8	76	97.4	412	99.5
douban	30	93.3	167	98.8	193	99.0	344	99.4
markor	7	42.9	53	35.9	77	40.3	286	73.8
suntimes	6	50.0	29	65.5	36	72.2	533	94.2
dartchecker	4	75.0	10	90.0	16	93.8	411	99.8
average	12	70.5	72	80.5	89	82.5	315	90.6

The results show that for all the subjects except *antennapod* and *markor*, at the most detailed content level, less than 20% of the exploration jumps outside the subject application. This suggests that cvrip is effective in preventing cross-app exploration. For an application like *antennapod*, there is still much cross-app exploration (34.5% at the content level) because *antennapod* has too many widgets which can lead to cross-app jumps. Although cvrip forces the exploration to return back to the original application after a cross-app jump, an external screen has already been reached after the jump. The cross-app exploration could be prevented if we knew whether each widget might lead to a cross-app jump before triggering actions on it. However, as discussed in Section 5.1.4, providing such fine-grained knowledge is too costly.

- **RQ1: Widget Detection**

Table 4 lists the superiority index of different widget detection methods. A high value means the corresponding method performs better on more subject applications. Table 5 also compares the GUICC coverage between the best and worst widget detection methods on GUI ripping. The results in these tables show that from the overall perspective, yolov5+ocr performs best on the subjects, uied+ocr does not show a competitive performance. The reason is that yolov5+ocr provides a good balance between detection precision and recall. UIED has high detection recall but low precision due to the reported non-executable widgets. Compared with the missing reports, sometimes false reports have more negative impacts on the GUI exploration efficiency.

Table 4 The superiority index of widget detection methods

widget detection	activity	layout	executable	content	time(m)
canny+ocr	4.625	3.750	2.875	2.125	73.1
yolov3+ocr	5.375	5.875	6.250	6.125	75.8
yolov5+ocr	6.625	6.000	7.250	5.750	69.1
uied+ocr	2.125	3.125	2.375	4.750	70.8

Table 5 Coverage under the best/worst widget detection strategies

app	yolov5+ocr				uied+ocr			
	activity	layout	exec.	content	activity	layout	exec.	content
trivago	24	113	138	408	18	46	53	173
antennapod	8	90	121	316	6	74	94	203
flipboard	29	259	297	454	24	237	264	425
runnerup	16	75	99	263	9	26	45	124
anymemo	21	91	110	285	15	32	40	239
amaze	6	49	79	375	5	46	76	412
douban	32	143	195	438	30	167	193	344
markor	15	52	81	392	7	53	77	286
suntimes	8	32	40	514	6	29	36	533
dartchecker	4	11	20	358	4	10	16	411
average	16	92	118	380	12	72	90	315

- **RQ2: State Recognition**

Table 6 presents the superiority index of different state recognition strategies. For the image comparison methods (*ssim*:0.5:0.9 vs. *ssim+orb*:0.5:0.9), SSIM and SSIM+ORB have close performance. SSIM covers more GUICC activities and contents, while SSIM+ORB covers more GUICC layouts and executables. For the screen resizing rates (*ssim*:0.5:0.9 vs. *ssim*:0.25:0.9), a resizing rate of 0.5 outperforms 0.25 in the high-level coverage but performs a little worse in the low-level coverage. Considering that the high-level coverage is often more important than the low-level coverage, resizing rate 0.5 works better for GUI ripping. For the image comparison thresholds (*ssim*:0.5:0.8 vs. *ssim*:0.5:0.9), a threshold of 0.8 can lead to better coverage at the first three GUICC levels. It is considered better than a threshold of 0.9. In summary, a state recognition strategy ssim:0.5:0.8 that keeps high image quality but decreases image comparison thresholds performs better for GUI riping. Its time cost is also close to other methods. Therefore, it is a recommended state recognition strategy.

Table 6 The superiority index of state recognition strategies

state recognition	activity	layout	exec.	content	time(m)
ssim:0.5:0.9	4.625	3.375	3.000	5.875	79.7
ssim:0.25:0.9	3.375	4.750	4.875	5.625	81.0
ssim_orb:0.5:0.9	4.250	6.000	5.500	3.625	76.2
ssim:0.5:0.8	6.500	4.625	5.375	3.625	78.5

- **RQ3: State Recognition Acceleration**

Table 7 lists the GUI ripping time to perform 1000 actions under different state recognition acceleration methods. Its data show that the non-accelerated ripping averagely needs 91m, while the accelerated ones take about 78 minutes. The acceleration reduces the analysis time by about 14%. The experiment cost about 1s per action to get the GUI hierarchy data for evaluation. That data is not necessary for the normal use of cvrip. If removing this GUI hierarchy acquiring time, the time reduction is about 17%. Although the GUI ripping is not significantly speeded up on some applications, we can see that the acceleration successfully prevents the GUI exploration time from exploding on applications like *antennapod* and *douban.* Without state recognition acceleration, the GUI ripping time may rapidly increase on some applications, while with the acceleration, that time can be kept relatively stable on different applications.

Table 7 GUI exploration time under different state recognition acceleration strategies

app	no acceleration	phash	dhash	ahash	resnet
trivago	84.8	77.5	77.9	70.9	80.8
antennapod	109.9	91.9	77.6	75.1	78.8
flipboard	69.1	81.5	81.6	83.9	91.3
runnerup	78.7	79.9	70.9	69.9	72.4
anymemo	77.4	75.8	70.9	71.0	72.2
amaze	73.7	76.4	74.1	69.6	71.3
douban	182.9	82.1	92.5	87.3	89.6
markor	82.5	69.5	81.0	69.2	75.0
suntimes	76.0	75.0	76.9	75.3	79.2
dartchecker	78.1	68.9	81.2	92.0	70.1
average	91.3	77.8	78.5	76.4	78.1

The abstract features used by the acceleration may not accurately distinguish GUI states. Consequently, different feature abstraction methods may have different impacts on the exploration coverage under limited action steps. Table 8 presents the superiority index of different state recognition acceleration strategies to evaluate such impacts. The higher the index value, the fewer negative impacts of the abstract-feature-based acceleration on the exploration coverage. The table data show that the non-accelerated methods can achieve more coverage at the GUICC activity level and relatively high coverage at the other GUICC levels. Considering the importance of the high-level coverages, the non-accelerated method has better exploration scope under limited action steps. Among the accelerated methods, the phash and dhash based methods have better performance, while the resnet-based method does not show advantages. That suggests even simple phash and dhash based methods are good choices for state recognition acceleration.

Table 8 The superiority index of different state recognition acceleration strategies

acceleration	activity	layout	exec.	content
none	4.875	4.000	3.500	2.875
phash1k	3.875	3.938	5.063	4.500
dhash1k	4.063	5.000	4.625	4.438
ahash1k	3.438	3.188	2.938	3.250
resnet50	3.125	3.250	3.250	4.313

- **RQ4: Widget Selection**

Table 9 lists the superiority index of different widget selection strategies for unclicked widgets on a GUI state in achieving exploration coverage. Its data show that the combined strategy outperforms the other strategies at all levels of GUICC coverage. The spatial diversity first strategy gets the second best performance, which means the spatial diversity benefits visual GUI ripping. The non-large-text-block-first strategy outperforms the random selection strategy at the first two GUICC levels, and the equivalence grouping strategy outperforms the random selection strategy at the later two GUICC levels. This suggests that the heuristics in these two strategies are also helpful to some extent. By combining many heuristics, an optimized GUI ripping efficiency can be achieved in the combined strategy.

Table 9 The superiority index of widget selection strategies

widget selection	activity	layout	exec.	content	time (m)
random selection	3.063	3.125	2.688	2.688	78.9
spatial diversity first	4.625	4.250	3.750	4.750	80.7
non-large-text-block-first	3.500	3.250	2.250	2.188	75.7
equivalence grouping	2.563	2.438	3.875	3.563	75.7
combined	5.625	6.3125	6.813	6.188	80.0

- **RQ5: Comparision with random CV-based exploration**

Table 10 shows the superiority index of a model-based GUI exploration method applying the best state abstraction strategy *ssim*:0.5:0.8 found in RQ2 (other configurations are set to default) and a method doing random model-less exploration on the visually detected widgets in achieving coverage after 1000 GUI actions, together with the exploration time. Table 11 is a detailed comparison of the achieved coverage on each subject application. The data in these tables show that the model-based exploration outperforms the random one at all GUICC coverage levels. This indicates that even under visual analysis, model-based GUI exploration is more efficient than a model-less approach

Table 10 The superiority index of a model-based GUI exploration and a random exploration

state recognition	activity	layout	exec.	content	time(m)
ssim:0.5:0.8	6	7	7	7	78.5
none	4	3	3	3	82.3

Table 11 Coverage under model-based and model-less exploration

App	ssim:0.5:0.8				no state recognition			
	activity	layout	exec.	content	activity	layout	exec.	content
trivago	22	66	79	193	19	65	87	174
antennapod	13	206	233	451	4	24	31	40
flipboard	25	262	286	422	28	265	282	411
runnerup	9	26	44	121	7	25	39	84
anymemo	23	46	62	287	7	25	39	84
amaze	15	53	80	318	6	9	12	17
douban	29	144	181	320	19	137	216	358
markor	14	45	66	210	9	37	54	199
suntimes	4	28	38	571	3	17	28	569
dartchecker	4	10	12	104	3	11	16	129
average	16	89	108	300	11	62	80	207

- **RQ6: Comparison with non-CV-based methods**

Regarding the GUI ripping time, cvrip takes about 75 minutes to trigger 1000 actions. Excluding 15m used to acquire GUI hierarchy data for evaluation purposes, the actual ripping time is about 60m. Fastbot and Ape take about 32m and 25m to trigger the same number of actions, respectively. This means, currently, the visual GUI ripping in cvrip runs slower than the traditional approaches in absolute time.

Table 12 presents the exploration coverage achieved by cvrip under the best strategy combination tested during the experiment (*yolov*5+*random*+*phash*+*ssim*:0.5:0.9) and by Fastbot and Ape in trigging the same number of GUI actions. The results show that cvrip covers more GUICC activities and contents than

Fastbot and Ape. Compared with Ape, cvrip covers fewer GUICC layouts and executables, but the gap is small. Considering the importance of the top-level coverage, cvrip has better per-GUI-action exploration efficiency on the subjects under test. This does not mean cvrip has overwhelming advantages over the two existing tools, but at least it shows that visual GUI ripping is a competitive choice.

Table 12 A comparison between the achieved coverage of cvrip, Fastbot, and Ape

App	cvrip				Fastbot				Ape			
	activity	layout	exec.	content	activity	layout	exec.	content	activity	layout	exec.	content
trivago	24	113	138	408	20	95	152	373	23	153	198	258
antennapod	8	90	121	316	3	86	135	228	5	110	169	336
flipboard	29	259	297	454	18	194	211	361	19	240	279	438
runnerup	16	75	99	263	7	49	156	369	5	40	86	299
anymemo	21	91	110	285	14	44	82	169	13	60	98	217
amaze	6	49	79	375	3	64	148	243	8	51	108	439
douban	32	143	195	438	25	142	181	349	25	130	163	434
markor	15	52	81	392	4	57	121	282	5	70	146	399
suntimes	8	32	40	514	9	79	133	402	8	88	142	390
dartchecker	4	11	20	358	9	23	139	458	8	30	53	435
average	16	92	118	380	11	83	146	323	12	97	144	365

6.3. Threats to Validity

There are three major validity threats in the experiments. (1) The first is that the number and types of the subject applications are limited. This is because the experiment is very costly. Each application requires about one day to run cvrip under all configurations, and the time may double if the *adb* in Android crash. Nevertheless, with these subjects covering many application domains, we believe it is possible to draw some preliminary conclusions. (2) The experiments only evaluate the exploration coverage from the GUI hierarchy perspective and do not check the source or byte code level coverage. Code coverage is a good reference metric. However, as discussed in Section 6.1, high coverage does not mean large exploration scope of the GUI screens. The coverage metric does not necessarily reflect the ability of a GUI ripping method. (3) The experiments fix the choice of other strategies when evaluating the impacts of a specific type of strategy. This is a common way of experimental study but may not cover some strategy combinations. We will continuously report findings on these strategies in the future use of cvrip.

7. Related Work

7.1. GUI Ripping

Great efforts have been devoted to GUI ripping [27], especially for Android GUI applications [34]. Some existing work uses GUI events as the states of the ripped GUI model (GUITAR [28], MobiGUITAR [1]). Most of the work abstract GUI states based on the implementation artifacts like Android activity or class of a GUI screen [2] and the runtime GUI structure information (e.g., [20][22][24],) and abstract state transitions from GUI actions. LATTE [39] integrates the back stack into the GUI model to express the back behavior in Android. Stoat [32] uses a stochastic Finite State Machine to model the application behavior. APE [17] dynamically adapts/changes the model abstraction to balance model precision and size based on runtime information. In the ways of GUI exploration, SwiftHand [9] introduces techniques to avoid application restarts as much as possible during GUI interactions. AIMDROID [18] implements an activity-insulated multi-level strategy to explore GUI applications. CrawlDroid [6] groups equivalent widgets in a state and applies a feedback-based exploration strategy. Recently, new techniques like snapshot taking [14] and reinforcement learning [29][31] have also been used to optimize GUI model exploration.

Compared with the existing work, a main characteristic of cvrip is that it does not depend on the underlying GUI framework to provide the GUI structure information. We only rely on the operating system to capture GUI screenshots and trigger GUI actions. Such functionalities are common in major operating systems and can be provided even without operating system support in robotic testing [30]. This makes the visual GUI ripping approach easily adaptable to various platforms.

7.2. Visual GUI Analysis

Recently, there has been much work on visual GUI analysis. For example, Bernal-Cárdenas et al. [5] analyze touch-indicator-enabled screen recording videos to extract replayable app usage scenarios. Cooper et al. [11] exploit the clues in screen images to detect duplicated bug reports. Xiao et al. [38] classify icon images to find security flaws.

There is also some work on problems close to visual GUI ripping. Humanoid [25] learns a model based on screen and widget images to predict how a human user will interact on a GUI screen, thereby enabling faster GUI testing. However, the tool needs the underlying GUI framework to provide GUI widget information and hence is not a pure visual solution. White et al. [36] detect widgets on Java Swing applications via YOLO v2 to conduct random testing. Deep GUI [12] builds a deep learning model to predict which screen coordinates can trigger valid GUI actions from screenshot images. Both two methods do random testing and do not construct GUI models. Our cvrip differs from them in that it can not only test an application but also produce a GUI model with states, widgets, and transitions which could be used in various software engining tasks.

8. Conclusion

This paper presents a visual GUI ripping framework together with experimental evaluations of various strategies supported by the framework. The experimental results show that, for the studied subject applications, although visual GUI ripping takes more time due to the computer vision algorithms, it can achieve more exploration coverage under the same number of triggered GUI actions than two state-of-the-art GUI ripping tools. This suggests that more future work is worthy of being conducted on GUI ripping in the visual way.

Acknowledgments

This work is supported by the Fundamental Research Funds for the Central Universities, NO. NS2021066.

REFERENCES

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, A. Memon. MobiGUITAR: Automated model-based testing of mobile apps. IEEE Software, 2014, 32(5):53-59

[2] Y. M. Baek, D. H. Bae. Automated model-based android GUI testing using multi-level GUI comparison criteria. In the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 238-249

[3] F. Behrang, S. P. Reiss, and A. Orso. GUIfetch: Supporting app design and development through GUI search. In the 5th ACM/IEEE International Conference on Mobile Software Engineering and Systems, 2018, pp. 236–246

[4] I. Banerjee, Advances in model-based testing of GUI-based software, Advances in Computers, 2017, vol. 105, pp. 45-78

[5] 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 International Conference on Software Engineering (ICSE), 2020

[6] Y. Cao, G. Wu, W. Chen, J. Wei. CrawlDroid: Effective model-based GUI testing of android apps. In Asia-Pacific Symposium on Internetware, 2018

[7] N. Chang, L. Wang, Y. Pei, S. K. Mondal, and X. Li. Change based test script maintenance for Android apps. In IEEE International Conference on Software Quality, Reliability and Security (QRS), 2018, pp. 215–225

[8] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, G.Li. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In Symposium on the Foundations of Software Engineering (FSE), 2020, pp. 1202-1214

[9] W. Choi, G. Necula, K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. ACM SIGPLAN Notices, 2013

[10] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet?. In the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 429–440

[11] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, D. Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In International Conference on Software Engineering (ICSE), 2021

[12] F. Y. B. Daragh and S. Malek. Deep GUI: Black-box GUI input generation with deep learning. In the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 905–916

[13] 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 the 30th Annual Symposium on User Interface Software and Technology (UIST), 2017

[14] Z. Dong, M. Böhme, L. Cojocaru, A. Roychoudhury. Time-travel testing of Android apps. In International Conference on Software Engineering (ICSE), 2020.

[15] S. R. Dubey. A decade survey of content based image retrieval using deep learning. IEEE Transactions on Circuits and Systems for Video Technology, 2022, 32(5): 2687–2704

[16] S. Feng, C. Chen. GIFdroid: Automated replay of visual bug reports for Android apps. In International Conference on Software Engineering (ICSE), 2022, pp. 1045–1057

[17] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, Z. Su. Practical GUI testing of Android applications via model abstraction and refinement. International Conference on Software Engineering (ICSE), 2019, pp. 269-280

[18] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, J. Lü. AIMDROID: Activity-insulated multi-level automated testing for android applications. In IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 103–114

[19] W. Guo, L. Shen, T. Su, X. Peng, and W. Xie. Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis. In IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 557–568

[20] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, R. Govindan. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In the 12th Annual International Conference on Mobile Systems, Applications, and Services, 2014, pp. 204–217

[21] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia. A survey on adaptive random testing. IEEE Transactions on Software Engineering, 2021, 47(10): 2052–2083

[22] B. Jiang, Y. Zhang, W. K. Chan, Z. Zhang. A systematic study on factors impacting GUI traversal-based test case generation techniques for Android applications. IEEE Transactions on Reliability, 2019, 68(3): 913–926

[23] C. Lee, S. Kim, D. Han, H. Yang, Y.-W. Park, B. C. Kwon, S. Ko. GUIComp: A GUI design assistant with real-time, multi-faceted feedback. In Proceedings of the CHI Conference on Human Factors in Computing Systems, 2020, pp. 1-13

[24] Y. Li, Z. Yang, Y. Guo, X. Chen. DroidBot: A lightweight UI-guided test input generator for Android. In IEEE/ACM 39th International Conference on Software Engineering, 2017, pp. 23–26

[25] Y. Li, Z. Yang, Y. Guo, X. Chen. Humanoid: A deep learning-based approach to automated black-box Android app testing. In IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019

[26] G. Li, G. Baechler, M. Tragut, Y. Li. Learning to denoise raw mobile UI layouts for improving datasets at scale. In CHI Conference on Human Factors in Computing Systems, 2022, pp. 1-13

[27] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins. The first decade of GUI ripping: Extensions, applications, and broader impacts. In 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 11–20

[28] B. N. Nguyen, B. Robbins, I. Banerjee, A. Memon. GUITAR: An innovative tool for automated testing of GUI-driven software. Automated Software Engineering, 2014

[29] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li. Reinforcement learning based curiosity-driven testing of Android applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2020, pp. 153–164

[30] J. Qian, Z. Shang, S. Yan, Y. Wang, L. Chen. RoScript: A visual script driven truly non-intrusive robotic testing system for touch screen applications. In International Conference on Software Engineering (ICSE), 2020, pp.297-308

[31] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. Deep reinforcement learning for black-box testing of android apps. ACM Transactions on Software Engineering and Methodology, 2022, 31(4): 1–29

[32] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su. Guided, stochastic model-based GUI testing of android apps. In ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), 2017

[33] R. Szeliski. Computer vision: algorithms and applications. Springer-Verlag, 2011

[34] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, T. Xie. An empirical study of android test generation tools in industrial cases. In the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 738–748, 2018.

[35] 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*, 2004, 13(4): 600–612

[36] T. D. White, G. Fraser, and G. J. Brown. Improving random GUI testing with image-based widget detection. In the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2019, pp.307-317

[37] W. Wang, W. Lam, and T. Xie. An infrastructure approach to improving effectiveness of Android UI testing tools. In the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2021, pp. 165–176

[38] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao. IconIntent: Automatic Identification of Sensitive UI Widgets Based on Icon Classification for Android Apps. In International Conference on Software Engineering (ICSE), 2019, pp. 257–268

[39] J. Yan, J. Yan, T. Wu, J. Zhang. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In IEEE International Conference on Software Quality, Reliability and Security, 2017

[40] PaddleOCR. https://github.com/PaddlePaddle/PaddleOCR

[41] cvrip. https://figshare.com/s/352020d939452019d138