GIFdroid: Automated Replay of Visual Bug Reports for Android Apps

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

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

ABSTRACT

Bug reports are vital for software maintenance that allow users to inform developers of the problems encountered while using software. However, it is difficult for non-technical users to write clear descriptions about the bug occurrence. Therefore, more and more users begin to record the screen for reporting bugs as it is easy to be created and contains detailed procedures triggering the bug. But it is still tedious and time-consuming for developers to reproduce the bug due to the length and unclear actions within the recording. To overcome these issues, we propose `GIFdroid`, a light-weight approach to automatically replay the execution trace from visual bug reports. `GIFdroid` adopts image processing techniques to extract the keyframes from the recording, map them to states in GUI Transitions Graph, and generate the execution trace of those states to trigger the bug. Our automated experiments and user study demonstrate its accuracy, efficiency, and usefulness of the approach.

CCS CONCEPTS

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

KEYWORDS

bug replay, visual recording, android testing

ACM Reference Format:
Sidong Feng and Chunyang Chen. 2022. GIFdroid: Automated Replay of Visual Bug Reports for Android Apps. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510048

1 INTRODUCTION

Software maintenance activities are known to be generally expensive and challenging [60] and one of the most important maintenance tasks is to handle bug reports [17]. A good bug report is detailed with clear information about what happened and what the user expected to happen. It goes on to contain a reproduction step or stack trace to assist developers in reproducing the bug, and supplement information such as screenshots, error logs, and environments. As long as this bug report is accurate, it should be straightforward for developers to reproduce and fix.

Bugs are often encountered by non-technical users who will document a description of the bug with steps to reproduce it. However, clear and concise bug reporting takes time, especially for non-developer or non-tester users who do not have that expertise and are not willing to spend that much effort [18, 25]. A poorly written report would be even poorly interpreted [24, 26, 36], which often devolves into the familiar repetitive back and forth and the need to nag that comes with every single bug. That effort may further prohibit users' contribution to bug reporting.

Compared to writing it down with instructions on how to replicate, video-based bug reports significantly lower the bar for documenting the bug. First, it is easy to record the screen as there are many tools available [3, 13], some of which are even embedded in the operating system by default like iOS [10] and Android [12]. Second, video recording can include more detail and context such as configurations, and parameters, hence it bridges the understanding gap between users and developers. That convenience may even better engage users in actively providing feedback to improve the app.

Despite the pros of the video-based bug report, it still requires developers to manually check each frame in the video and repeat it in their environment. According to our empirical study of 13,587 bug recordings from 647 Android apps in Section 2, one video is of 148.29 frames on average with a varied resolution for manual observation. In addition, only 6.8% of video recordings start from the app launch and most recordings begin 2-7 steps before the bug occurrence, indicating that developers need to guess steps to the entry frame of the video by themselves. Therefore, it is necessary to develop an automated bug replay tool from video-based bug reports to save developers' effort in a bug fix.

There are many related works on bug replay but rarely related to visual bug reports. Some researchers [37, 65, 70, 83, 84] leverage the natural language processing methods with program analysis to generate the test cases from the textual descriptions in bug reports. However, those approaches do not apply to video-based bug reports. There are platforms providing both video recording and replaying functionalities [1, 23, 45, 62] which also store the low-level program execution information. They require the framework installation or app instrumentation which is too heavy for end users. Users tend to use general recording tools to get the video that only contains the visual information, according to our observation in Section 2.2. Automation for processing general recordings to reproduce bugs is necessary and would help developers shift their focus toward bug fixing.

*Corresponding authors

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510048

To automate the analysis of bug recordings, we introduce GIFdroid, a light-weight image-processing approach to automatically replay the video (GIF) based bug reports for Android apps. First, we extract keyframes (i.e., fully rendered GUIs) of a recording by comparing the similarity of consecutive frames. Second, a sequence of located keyframes is then mapped to the GUI states in the existing UTG (UI Transition Graph) of the app by calculating image similarity based on pixel and structural features. Third, given the mapped sequence, we propose a novel algorithm to not only address the defective mapped sequence, but also auto-complete the missing trace between app launch to the entry frame of the video, resulting in an optimal execution trace to automatically repeat the bug trigger.

To evaluate the effectiveness of our tool, we create the groudtruth by manually labelling 61 video recordings from 31 apps. As our approach consists of three main components, we evaluate them one by one including keyframe location, GUI mapping, and trace generation. In all tasks, our approach significantly outperforms other baselines and successfully reproduce 82% video recordings. Apart from the accuracy of our tool, we also evaluate the usefulness of our tool by conducting a user study on replaying bugs from 10 real-world video recordings in GitHub. Through the study, we provide the initial evidence of the usefulness of GIFdroid for bootstrapping bug replay.

The contributions of this paper are as follows:

- The first light-weight image-processing based approach, GIFdroid, to reproduce bugs for Android apps directly from the GIF recordings with code released for public[1].
- A motivational empirical study of large-scale recordings within real-world issue reports from GitHub including their content, length, etc.
- A comprehensive evaluation including automated experiments and a user study to demonstrate the accuracy, efficiency and usefulness of our approach.

2 MOTIVATIONAL MINING STUDY

While the main focus and contribution of this work is developing an approach to automatically replay the visual bug reports, we still carry out an empirical study to understand the characteristics of visual bug reports. The overall status of visual bug reports can clearly frame the context and motivation of this work and their characteristics will be taken into consideration in our approach design. But note that this motivational study just aims to provide an initial analysis towards developers supporting visual bug reports, and a more comprehensive empirical study would be needed to deeply understand it.

2.1 Data Collection

We choose F-droid [5] as the source of our study subjects, as it contains a large set of apps (1,274 at the time of our study) covering diverse categories such as connectivity, financial, multimedia. All apps are open-source hosted on platforms like GitHub [8] which makes it possible for us to access their issue repositories for analysis.

We built a web crawler to automatically crawl the visual bug reports from issue repositories of these apps containing visual recordings (e.g., animation, video) with suffix names like .gif, .mp4, etc., or URL from video sharing platforms (e.g., GIPHY[2] [7], YouTube [15]). It took us two weeks to scan 408,272 issues from 1,274 apps and finally mined 13,587 visual recordings from 7,230 issues (647 apps). The dataset consists of 8,698 GIFs and 4,889 videos. As shown in Figure 1, attaching visual recording in the issue report is more and more popular. Within the video, there is always some extra information (e.g., caption, whole screen of the PC, face of speaker) in addition to the app usage screen. Since there are more GIFs than videos, we focus on the GIF visual recording for brevity in this paper.

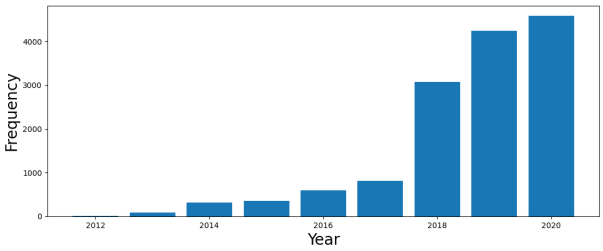

Figure 1: Number of recordings in GitHub from 2012 to 2020

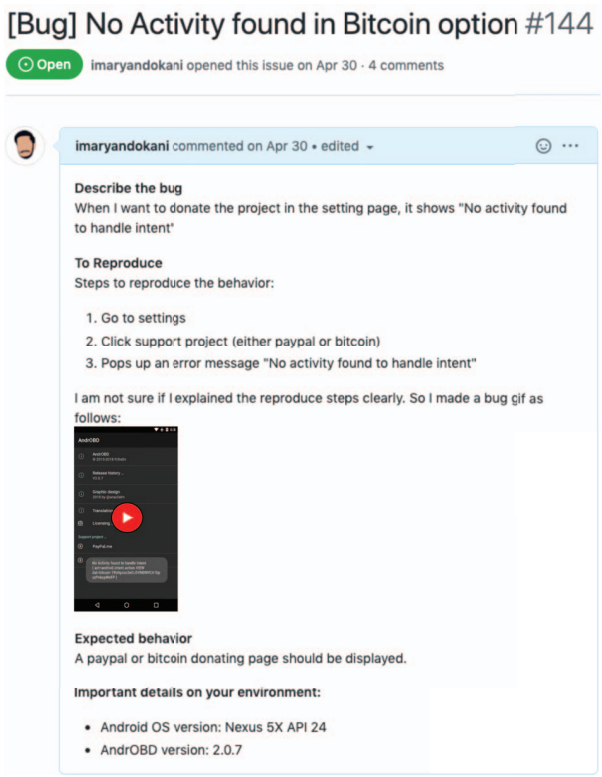

Figure 2: Example of the issue reports for Bug Replay. The reporters provide a visual recording (i.e., gif) to help developers understand the bug report more clearly.

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

[2]GIPHY is the most popular animated GIF sharing platform, serving 700 million users

Table 1: Recording resolution.

Ratios	Occurrences	Resolution
16:9	288 (48%)	1920×1080
		1280×720
		960×540
5:3	63 (10.5%)	1280×800
		800×480
3:2	21 (3.5%)	900×600
		480×320

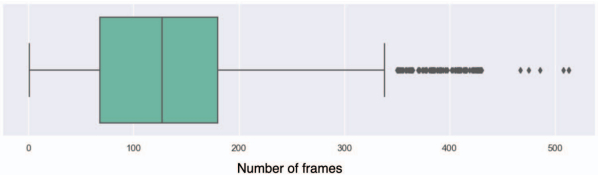

Figure 3: Number of frames of each recording.

2.2 What are in Visual Recordings?

To understand the content within these visual recordings, the first two authors manually check 1,000 (11.5%) GIFs and their corresponding descriptions in the issue report. Following the Card Sorting [67] method, we classify those GIFs into three categories:

(i) Bug Replay (60%). Most screen recordings are about the bug replay including detailed procedures in triggering the bugs, especially for those requiring complicated actions with one example seen in Figure 2[3]. The visual recordings bridge the lexical gap between users and developers and help developers read and comprehend the bug report reasoning about the problem from the high-level description.

(ii) Issue Fixed (17%). Once the issue is fixed, developers tend to record the screen to display the update changes, along with a checklist for project owners to approve the pull request, Also, those recordings are used for app introduction or answering users' feature requests.

(iii) Feature Request (23%). Users may use recordings to ask for missing functionality (e.g., provided by other apps) or missing content (e.g., in catalogues and games), and sharing ideas on how to improve the app in future releases by adding or changing features.

2.3 What are Characteristics of Bug Recordings?

Since the target of this work is to auto-replay the bug recording, we further analyze the characteristics of 600 bug recordings classified in Section 2.2. Due to the difference of devices or users' resize of the video, there may be different resolutions and aspect ratios as summarized in Table 1. About half of bug recordings are of 16:9 aspect ratio with varied resolution from high-quality 1920x1080 to 960x540. There are also some minor aspect ratios and resolutions as users may post-process the recording by resizing or cropping arbitrarily, according to our observation.

Figure 3 depicts the number of frames in visual bug recording. On average, there are 148.29 frames per video and some of them are even with more than 500 frames which require developers to manually check and replay in their own settings. We also find that only 41 recordings (6.8%) start from the launch of the app, while most recordings start from 2-7 steps before the bug occurrence. It indicates that there is no explicit hint to guide developers to the entry frame of the recording and that barrier may negatively influence developers' efficiency. As the important information to show users' actions, some recordings contain touch indicators (e.g., circle or arrow of the touch position) which is an important resource to replay the bug [23]. However, there are only 155 recordings (25.8%) enabling the "Show Touches" option in our dataset. Developers especially novice ones have to guess and try the potential actions to trigger the target page from the current page.

Summary: By analyzing issue reports from 1,274 existing apps crawled from F-Droid, 60% of them are with recordings for bug replay. A large number of frames and varied resolution make it difficult for developers to replay them in their setting. Such phenomenon is further exacerbated as 74.2% of recordings are without touch indicators on the screen. These findings confirm the necessity and difficulty of the replay of visual bug reports, and motivate our approach development for automatic replaying the recordings for developers and testers.

3 APPROACH

Given an input bug recording, we propose an automated approach to localize a sequence of keyframes in the GIF and subsequently map them to the existing UTG (UI Transition Graph) to extract the execution trace. The overview of our approach is shown in Figure 4, which is divided into three main phases: (i) the *Keyframe Location* phase, which identifies a sequence of keyframes of an input visual recording, (ii) the *GUI Mapping* phase that maps each located keyframe to the GUIs in UTG, yielding an index sequence, and (iii) the *Execution Trace Generation* phase that utilizes the index sequence to detect an optimal replayable execution trace.

3.1 Keyframe Location

Note that GUI rendering takes time, hence many frames in the visual recording are showing the partial rendering process. The goal of this phase is to locate keyframes i.e., states in which GUI are fully rendered in a given visual recording.

3.1.1 Consecutive Frame Comparison. Inspired by signal processing, we leverage the image processing techniques 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 [28, 68]. Y-Diff is the difference in Y (luminance) values of two images in the YUV color space. We adopt the luminance component as the human perception (a.k.a human vision) is sensitive to brightness changes. In the human visual system, a majority of visual information is conveyed by patterns of

[3]https://github.com/fr3ts0n/AndrOBD/issues/144

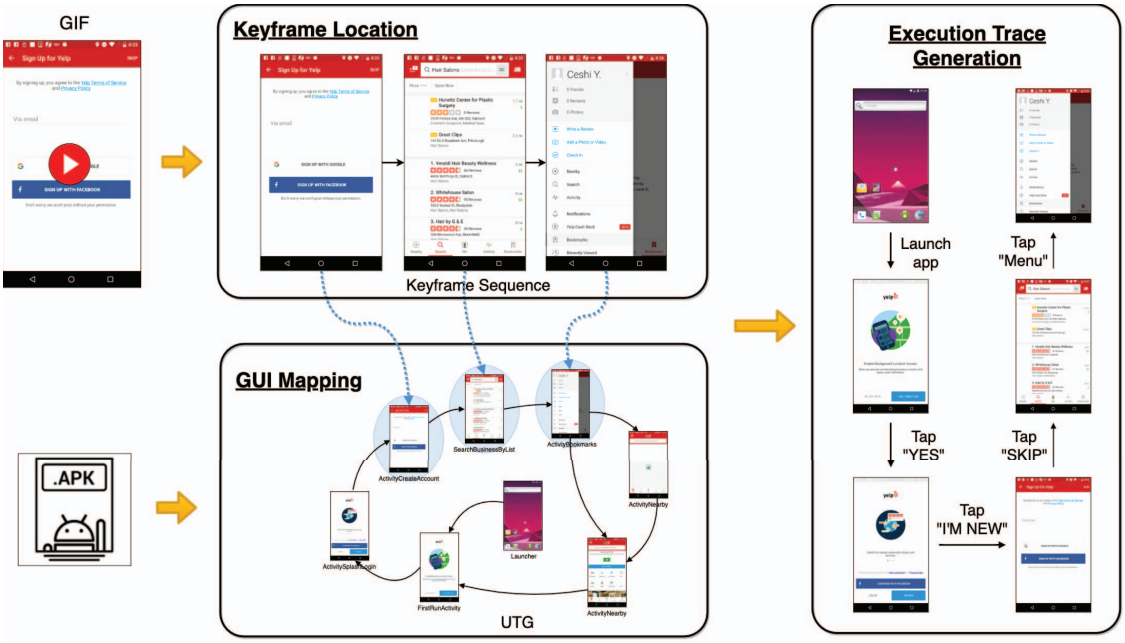

Figure 4: The overview of GIFdroid.

contrasts from its brightness changes [79]. Furthermore, luminance is a major input for the human perception of motion [50]. Since people perceive a sequence of graphics changes as a motion, consecutive images are perceptually similar if people do not recognize any motions from the image frames.

Consider a visual recording $\{f_0, f_1, .., f_{N-1}, f_N\}$, 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, SSIM (Structural Similarity Index) [73], to produce a per-pixel similarity value related to the local difference in the average value, the variance, and the correlation of luminances. In detail, the SSIM similarity for two luminance masks is defined as:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1) + (2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \quad (1)$$

where x, y denote the luminance masks Y_{N-1}, Y_N, and μ_x, σ_x, σ_{xy} are the mean, standard deviation, and cross correlation between the images, respectively. C_1 and C_2 are used to avoid instability when the means and variances are close to zero. A SSIM score is a number between 0 and 1, and a higher value indicates a strong level of similarity.

3.1.2 Keyframe Identification. To make decisions on whether the frame is a keyframe, we look into the similarity scores of consecutive frames in the visual recording as shown in Figure 5. The first step is to group frames belonging to the same atomic activity according to a tailored pattern analysis. This procedure is necessary because discrete activities performed on the screen will persist

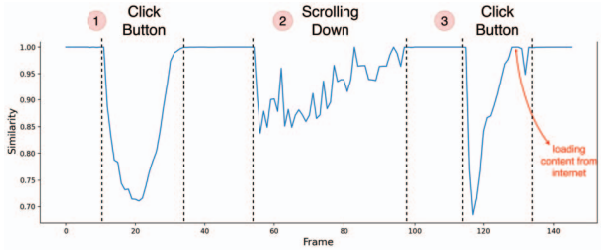

Figure 5: An illustration of the Y-Diff similarity scores of consecutive frames in the visual recording.

across several frames, and thus, need to be grouped and segmented accordingly. There are 3 types of patterns, i.e., *instantaneous transitions*, *animation transitions*, and *steady*.

(1) instantaneous transitions: As shown in Figure 5 Activity 1 (clicking a button), the similarity score starts to drop drastically which reveals an instantaneous transition from one screen to another. In addition, one common case is that the similarity score becomes steady for a small period of time t_s between two drastically droppings as shown in Figure 5 Activity 3. The occurrence of this short steady duration t_s is because GUI has not finished loading. While the GUI layout of GUI rendering is fast, resources loading may take time. For example, rendering images from the web depends on device bandwidth, image loading efficiency, etc.

(2) animation transitions: Figure 5 Activity 2 shows the similarity sequence of the transitions where animation effects are used, for example, the "scrolling" event. That is, over a period of time, the similarity score continues to increase slightly.

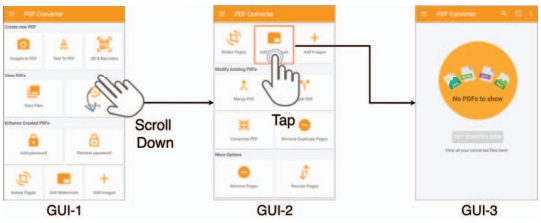

Figure 6: An example of partial UTG on mobile app GUI proceeding from state GUI-1 to state GUI-3.

(3) steady: Figure 5 gives an example of a GUI steady state where the consecutive frames are similar for a relatively long duration. We have empirically set 0.95[4] as the threshold to decide whether two frames are similar, and 5 frames as the threshold to indicate a steady state.

3.2 GUI Mapping

It is easy for developers to get the UTG of their own app [31, 77]. Therefore, instead of inferring actions from the recording [23, 80], we directly map keyframes extracted from the recording to states/GUIs within the UTG. To achieve this, we compute the image similarity between the keyframe and each GUI screenshot based on both pixel and structural features. The GUI screenshot that has the highest similarity score is regarded as the index of the keyframe.

3.2.1 UTG Construction. A GUI transitions graph (UTG) of an Android app is widely used to illustrate the transitions across different GUIs triggered by typical elements such as toasts (pop-ups), text boxes, text view objects, spinners, list items, progress bars, checkboxes. In Figure 6 we illustrate how a UTG emerges as a result of user interactions in an app. Starting with the GUI-1, the user "scrolls down" to the bottom of the page (GUI-2). When the user "taps" a button, the GUI transits to the GUI-3. There are many tools to construct a UTG, either manually [11] or automatically [19, 48]. In this paper, we adopt the Firebase [6], a widely-used automated GUI exploration tool developed by Google, while other tools can also be used.

3.2.2 Feature Extraction. The basic need for any image searching techniques is to detect and construct a discriminative feature representation [34, 35]. A proper construction of features can improve the performance of the image searching method [47]. According to our observation, in addition to the pixel features as that in natural images, GUI screenshots are also of additional structural features i.e., the layout of different components in one page. Therefore, we adopt a hybrid method based on two types of features, SSIM (Structural Similarity Index) [73] and ORB (Oriented FAST and Rotated BRIEF) [63], for searching the mapping GUI screenshots in the UTG for the keyframe.

While SSIM detects the features within pixels and structures (i.e., a detailed description is demonstrated in Section 3.1.1), it still has several fundamental limitations that exist in visual recordings, e.g., image distortion [46, 71]. To address this, we further supplement a local invariant feature extraction method, ORB. Given an image,

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

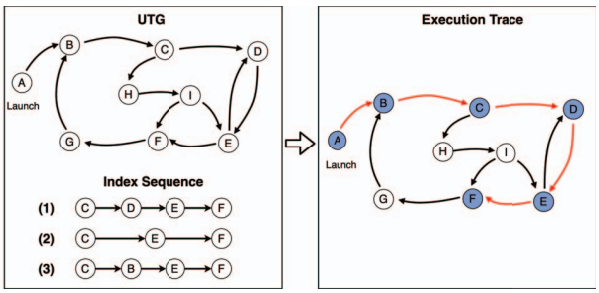

Figure 7: Illustration of the execution trace generation. Index sequence 2,3 indicate two types of defective sequences, i.e., missing $\{D\}$ and wrong mapping to $\{B\}$, respectively.

ORB first detects the interest points, indicating at which the direction of the boundary of the object changes abruptly or intersection point between two or more edge segments. Then, ORB converts each interest point into an n-bits binary feature descriptor, which acts as a "fingerprint" that can be used to differentiate one feature from another. A feature descriptor of an interest point is computed by an intensity difference test τ:

$$\tau(p; x, y) = \begin{cases} 1 & \text{if p(x)} < \text{p(y)} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

where $p(x), p(y)$ are intensity values at pixel x, y around the interest point. Due to the characteristic of local feature extraction, ORB features remain invariant of scale, brightness, and also maintain a certain stability of affine transformation, and noise.

3.2.3 Similarity Computation. Based on the features extracted by SSIM and ORB, we compute a similarity value S_{ssim} and S_{orb}, respectively. To compute S_{orb}, we adopt the Brute Force algorithm [63], which compares the hamming distance between feature descriptors. We compute the S_{ssim} using Equation 1 based on similarity on luminance, contrast, and structure. We then further determine the similarity S_{comb} between the keyframe and states in UTG by combining two feature similarities score:

$$S_{comb} = w \times S_{orb} + (1 - w) \times S_{ssim} \quad (3)$$

where w is a weight for S_{ssim} and S_{orb}, taking a value between 0 to 1. Smaller w value weights S_{ssim} more heavily, and larger value weights S_{orb} more heavily. We empirically choose 0.5 as the w value for the best performance.

Based on the combined similarity between the keyframe and each GUI screenshot, we select the highest score to be the index of the keyframe. Consequently, a sequence of keyframes is converted to a sequence of the index in the UTG.

3.3 Execution Trace Generation

After mapping keyframes to the GUIs in the UTG, we need to go one step further to connect these GUIs/states into a trace to replay the bug. However, this process is challenging due to two reasons. First, the extracted keyframe (Section 3.1) and mapped GUIs (Section 3.2) may not be 100% accurate, resulting in a mismatch of the groundtruth trace. For example in Figure 7, $\{D\}$ is missed in the index sequence 2 and the second keyframe is wrongly mapping

to $\{B\}$ in the index sequence 3. Second, different from the uploaded GIF which may start the recording anytime, the recovered trace in our case must begin from the launch of the app.

Therefore, the trace generation algorithm needs to consider both the wrong extraction/mapping in our previous steps, and the missing trace between the app launch and first keyframe in the visual bug report. To overcome these issues, our approach first generates all candidate sequences in UTG between the app launch to the last keyframe from GIF. By regarding the extracted keyframes as a sequence, our approach then further extracts the Longest Common Subsequence (LCS) between it and all candidate sequences.

The overflow of our approach can be seen in Algorithm 1. Given an index sequence $X = \{x_1, x_2, ..., x_n\}$ (extracted by keyframe mapping) where x_n is the last node, a UTG graph G, and an app launch node s. To find acyclic paths (avoiding dead loops) from app launch node (s) to last index node (x_n), we adopt Depth-First Search traversal, that takes a path on G and starts walking on it and check if it reaches the destination then count the path and backtrack to take another path. To avoid cyclic path, we record all visited nodes (Line 5), so that one node cannot be visited twice. The output of the traversal is a set of acyclic path $SEQ = \{y_1, y_2, ..., y_m\}$ where $y_1 = s$ and $y_m = x_n$. We regard those acyclic paths as the candidate sequences for execution. For example, in Figure 7, there are three candidate sequences from launch node $\{A\}$ to last index node $\{F\}$ are found (1)$\{A, B, C, D, E, F\}$, (2)$\{A, B, C, H, I, F\}$, (3)$\{A, B, C, H, I, E, F\}$. Note that $\{A, B, C, D, E, D, E, F\}$ is omitted due to cyclic.

Next, for each candidate sequence SEQ, we adopt the dynamic programming algorithm to find the LCS to the index sequence X (Lines 17-32), in respect to detect how many index nodes are covered in this candidate sequence. The recursive solution to detect the LCS for each candidate sequence SEQ to index sequence X can be defined as:

$$com[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \text{com[i-1,j-1]+1} & \text{if } x[i] = seq[j] \\ \text{max(com[i,j-1],com[i-1,j])} & \text{if } x[i] \neq seq[j] \end{cases} \quad (4)$$

where com to be a table for X and SEQ (e.g., $com[i, j]$ is the relation between X_i and SEQ_j). A LCS can be found by tracing the table com. We omit the details of tracing the table as $FindLCS(com)$ due to space limitations. Note that we initialize first row and column of com as zero to prevent the occurrence of NULL (Line 19). For example, in Figure 7, the LCSs between index sequence and three candidate sequences are (1)$\{C, E, F\}$ (2)$\{C, F\}$ (3)$\{C, E, F\}$, respectively.

Once the LCSs are detected, we select the candidate sequence that has the longest LCS as the execution trace due to it replays most keyframes (or index nodes) in the visual recording. Besides, our goal is to help developers reproduce the bug with the least amount of time/steps. Therefore, we choose the optimal execution trace with the shortest sequence. For example, in Figure 7, (2)$\{A, B, C, H, I, F\}$ is omitted due to it does not replay the most index nodes, (i.e., its LCS $\{C, F\}$ is not the longest). (3)$\{A, B, C, H, I, E, F\}$ is omitted due to it is not the optimal trace (i.e., not the shortest sequence). Therefore, the optimal execution trace (1)$\{A, B, C, D, E, F\}$ is generated based on the index sequence, even defective.

Algorithm 1: Execution Trace Generation

```
Inputs : X: index sequence {x_1, x_2, x_3, ..., x_n};
         G: UTG graph;
         s: starting node in UTG;
Output: execution trace
/* Find the candidate sequences (SEQs) from app launch node
   to last index node */
1  SEQs ← [] ;
2  SEQ ← [] ;
3  visited ← false for all nodes in UTG;
4  DFS (s, x_n, visited, SEQ)
5      visited[s] ← true ;            // prevent acyclic sequence
6      SEQ.append(s) ;
7      if s == x_n then
8          SEQs.append(SEQ) ;
9      else
10         foreach v ∈ G.Adj[s] do
11             if visited[v] == false then
12                 DFS (v, x_n, visied, SEQ)
13         end
14     end
       /* Backtrack to take another path */
15     SEQ.pop() ;
16     visited[s] ← false ;
   /* Find the LCSs between index sequence and each candidate
      sequence */
17 foreach SEQ ∈ SEQs do
18     let com[0..m, 0..n] be new table ;
19     initialize com ← 0 ;           // prevent NULL sequence
20     for i ← 1 to X.length do
21         for j ← 1 to SEQ.length do
22             if x[i] == SEQ[j] then
23                 com[i, j] ← com[i − 1, j − 1] + 1
24             else if com[i − 1, j] ≥ com[i, j − 1] then
25                 com[i, j] ← com[i − 1, j]
26             else
27                 com[i, j] ← com[i, j − 1]
28             end
29         end
30     end
31     LCSs ← save (FindLCS(com))
32 end
   /* Find the execution trace with longest LCSs and shortest
      sequences */
33 Trace ← SEQs [max(LCSs) & min(SEQs)] ;
34 return Trace
```

4 AUTOMATED EVALUATION OF GIFDROID

In this section, we describe the procedure we used to evaluate GIFdroid in terms of its performance automatically. We manually construct a dataset as the groundtruth for evaluating each step within our approach, instead of using the real-world bug recordings due to two reasons. First, many real-world bug reports have

been fixed and the app is also patched, but it is hard to find the corresponding previous version of the app for reproduction. Second, the replay of some bug reports (e.g., financial, social apps) requires much information like authentication/database/hardware to generate the UTG which are beyond the scope of this study. Therefore, we collect 61 visual recordings from 31 open-source Android apps which were used in previous studies [23, 26, 55]. They are also top-rated on Google Play covering 14 app categories (e.g., development, productivity, etc.). To make these recordings as similar to real-world bug reports as possible, we adopt different ways for generating the recording including different creation tools (32 from video conversion, 22 from mobile apps, 7 from emulator screen recording), varied resolutions (27 1920 × 1080, 23 1280 × 800, 11 900 × 600), diverse length (30-305 frames), and differed playing speed (7-30 frames per second).

For each app, we also collect its UTG by Google Firebase. Since our approach consists of three main steps, we evaluate each phase of `GIFdroid`, including Keyframe Location (Section 3.1), GUI Mapping (Section 3.2), and Execution Trace Generation (Section 3.3). Therefore, we ask two experienced developers to manually label keyframes from recordings, GUI mapping between recording and UTG, and real trace in the UTG as the groundtruth for each phase. Note that each human annotator finished the labelling individually and they discussed the difference until an agreement was reached.

4.1 Accuracy of Keyframe Location

Ground Truth: To evaluate the ability of keyframe location to accurately identify the keyframes present in the visual recordings, we manually generated a ground truth for the keyframe. We recruited two paid annotators from online posting who have experience in bug replay. To help ensure the validity and consistency of the ground truth, we first asked them to spend twenty minutes reading through the video annotation guidelines (i.e., TRECVID [58]) and get familiar with the widely-used annotation tool VirtualDub [14]. Then, we assigned the set of visual recordings to them to annotate keyframes independently without any discussion. After the annotation, the annotators met and sanity corrected the subtle discrepancies (i.e., ± 3 frames). Any disagreement would be handed over to one author for the final decision. In total, we obtained 289 keyframes for 61 recordings, 4.73 per recording on average, tallying with our observations for real-world recordings in Section 2.3.

Metrics: We employ three evaluation metrics, i.e., precision, recall, F1-score, to evaluate the performance of keyframe location. The predicted frame which lies in the ground truth interval is regarded as correctly predicted. Since there should be only one keyframe per interval, if two or more keyframes are localized in a single interval, only the first keyframe is counted as correct. Precision is the proportion of frames that are correctly predicted as keyframes among all frames predicted as keyframes.

$$precision = \frac{\#Frames\ correctly\ predicted\ as\ keyframes}{\#All\ frames\ predicted\ as\ keyframes}$$

Recall is the proportion of frames that are correctly predicted as keyframes among all keyframes.

$$recall = \frac{\#Frames\ correctly\ predicted\ as\ keyframes}{\#All\ keyframes}$$

Table 2: Performance comparison for keyframe location.

Method	Precision	Recall	F1-score
Hecate [66]	0.174	0.247	0.204
Comixify [59]	0.244	0.516	0.332
ILS-SUMM [64]	0.255	0.685	0.371
PySceneDetect [9]	0.418	0.550	0.475
GIFdroid	**0.858**	**0.904**	**0.880**

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.

Baselines: We set up four state-of-the-art methods which are widely used for keyframe extraction as the baselines to compare with our method. *Comixify* [59] is an unsupervised reinforcement learning method to predict for each frame a probability to extract keyframes in the videos. *ILS-SUMM* [64] formulates the keyframe extraction as an optimization problem, using a meta-heuristic optimization framework to extract a sequence of keyframes that has minimum feature distance. *Hecate* [66] is a tool developed by Yahoo that estimates frame quality on the image aesthetics and clusters them to select the centroid as a keyframe. *PySceneDetect* [9] is a practical tool implemented as a Python library that is popular in GitHub. The core technique for PySceneDetect to detect keyframes is a content-aware detection by analyzing the color, intensity, motion between frames.

Results: Table 2 shows the performance of all approaches. The performance of our method is much better than that of other baselines, i.e., 32%, 106%, 14% boost in recall, precision, and F1-score compared with the best baseline (ILS-SUMM, PySceneDetect). The issues with these baselines are that they are designed for general videos which contain more natural scenes like human, plants, animals etc. However, different from those videos, our visual bug recordings belong to artificial artifacts with different rendering processes. Therefore, considering the characteristics of visual bug recordings, our approach can work well in extracting keyframes.

Our approach also makes mistakes in keyframe extraction due to three reasons. First, within some apps, the resource loading is so slow that the partial GUI may stay for a relatively long time, beyond our threshold setting in Section 3.1. So, that frame is wrongly predicted as a keyframe. Second, some users may click the button before the GUI is fully rendered to the next page. That short period makes our approach miss the keyframe. Third, some GUIs may contain animated app elements such as advertisement or movie playing, which will change dynamically, resulting in no steady keyframes being localized.

4.2 Accuracy of GUI Mapping

Ground Truth: To evaluate the ability of GUI mapping to accurately search the nearest GUI screenshot in the UTG, we first collected the UTG and their GUI screenshots for those apps, following the procedure outlined in Section 3.2.1. Given the keyframes (i.e., a frame from the ground truth interval), we selected the most similar

Figure 8: Examples of bad cases in GUI mapping.

GUI screenshot in the UTG as the ground truth for GUI mapping. Note that labelling was finished by two annotators independently without any discussion, and any disagreement would be handed over to one author for the final decision. Thus, we obtained 289 ground truth pairs of keyframes and GUI screenshots.

Metrics: We formulate the problem of GUI mapping as an image searching task (i.e., search the most similar GUI screenshot), so we adopt Precision@k to evaluate the performance of GUI mapping. The higher value of the metric is, the better a search method performs. Precision@k is the proportion of the top-k results for a query GUI that contains the grouptruth one. Note we only consider k in the range 1-3, as developers rarely check a long recommendation list.

Baselines: To further demonstrate the advantage of our method, we compare it with 10 image processing baselines, including pixel level (e.g, euclidean distance [33], color histogram [72], fingerprint [16]), and structural level (e.g., SSIM [73], SIFT [52], SURF [21], ORB [63]).

Results: Table 3 shows the overall performance of all methods. In contrast with baselines, our method outperforms in all metrics, 85.4%, 90.0%, 91.3% for Precision@1, Precision@2, Precision@3 respectively. We observe that the methods based on structural features perform much better than pixel features due to the reason that the pixel similarity suffers from the scale-invariant as the resolutions for visual recordings varies. Our method that combines SSIM and ORB leads to a substantial improvement (i.e., 9.7% higher) over any single feature, indicating that they complement each other. In detail, ORB addresses the image distortion that causes false GUI mapping considering only SSIM.

Albeit the good performance of our method, we still make wrong mapping for some keyframes. We manually check those wrong mapping cases, and find that one common cause is the dynamically loaded content. For example, as seen in Figure 8, some GUIs mappings look visually very different as the pop-up window or images loaded from the internet may vary a lot at different time. As the dynamic content takes over a large area of the whole GUI, our approach based on visual features cannot accurately locate them.

4.3 Performance of Trace Generation

Ground Truth: To evaluate the ability of trace generation to accurately generate the replayable execution trace, we label the execution trace that can replay the visual recording from the app launch as the ground truth. Note that there may be multiple ground truth traces complying with the optimal definition (i.e., the shortest trace to replay the recording) in Section 3.3. Therefore, we manually check and label all the ground truths by two annotators independently without any discussion. To ensure the quality, any disagreement would be handed over to one author for the final decision. Totally, we obtained 61 execution traces, including 539 reproduction steps.

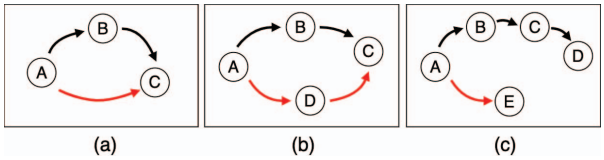

Figure 9: Illustration of bad case for trace generation. Black line represents ground truth. Red line represents the generated execution trace.

Metrics: To measure the similarity of the groundtruth trace and predicted replay sequence, we first get the LCS between two sequences. We then calculate the similarity [75] following $\frac{2\times M}{T}$ where M is the length of LCS, and T is the length of the sum of both sequences. It gives a real value with a range [0,1] and is usually expressed as a percentage. The higher the metric score, the more similar the generated execution trace is to the ground truth. If the generated trace exactly matches the ground truth, the similarity value is 1 (100%).

Baselines: We set up one state-of-the-art scenario generation method (V2S) and an ablation study of GIFdroid without LCS algorithm (GIFdroid-LCS) as our baselines. *V2S* [23] is the first purely graphical Android record-and-replay technique. V2S adopts deep learning models to detect user actions via classifying touch indicators for each frame in a video, and converts these into a script to replay the scenarios. Note that V2S has strict requirements about the input video including high resolution, frames per second and touch indicator for inferring detailed actions. Therefore, in addition to testing V2S in all our datasets, we also test its performance in the part of our dataset (i.e., 19 recordings) which contain touch indicators called *V2S+Touch*. To test the importance of the proposed algorithm that addresses defective index sequence in Section 3.3, we also add another ablation study called *GIFdroid-LCS* which does not contain that component.

Results: Table 4 shows the performance comparison with the baselines. Our method achieves 89.59% sequence similarity which is much higher than that of baselines. Note that due to the strict requirement of input recordings, V2S does not work well in all our datasets, but performs well in our partial dataset with touch indicators. Even for recordings with touch indicators, the extracted trace is still not that accurate as it could not recover the trace from the app launch to the entry in the recording. In a word, the hard requirements still limit its generality in the real testing environment especially those open-source software development. In addition, adding LCS can mitigate the errors introduced in the first two steps in our approach, resulting in a boost of performance from 82.63% to 89.59%. Although applying LCS takes a bit more runtime (i.e., 13.25 seconds on average), it does not influence its real-world usage as it can be automatically run offline.

Table 3: Performance comparison for GUI Mapping

Metric	Pixel					Stucture				GIFdroid
	Euclidean	Histogram	pHash	dHash	aHash	SSIM	SIFT	SURF	ORB	SSIM+ORB
Precision@1	0.0%	30.6%	46.6%	50.6%	38.6%	75.7%	63.6%	67.8%	75.7%	**85.4%**
Precision@2	1.3%	47.7%	58.6%	67.8%	61.3%	81.3%	81.3%	77.3%	83.7%	**90.0%**
Precision@3	1.3%	55.7%	63.6%	75.7%	66.6%	85.3%	89.3%	82.6%	89.3%	**91.3%**

Table 4: Performance comparison of execution generation.

Metric	V2S	V2S+Touch	GIFdroid-LCS	GIFdroid
Similarity	7.17%	63.19%	82.63%	**89.59%**
Time (sec)	791.23	856.91	**84.66**	97.91

Table 5 shows detailed results of the success rate for each visual recordings, where each app, execution trace (number of steps), and successfully replayed steps are displayed. Green cells indicate a fully reproduced execution trace from the video recording, Orange cells indicate more than 50% of steps reproduced, and Red cells indicate less than 50% of reproduced steps. GIFdroid fully reproduces 82% (50/61) of the visual recordings, signals a strong replay-ability. We manually check the instances where our method failed to reproduce scenarios. Instances where slightly biased (orange cells) are largely due to inaccuracies in keyframe location (i.e., in Figure 9(a), $\{B\}$ is missing) and GUI mapping (i.e., in Figure 9(b), $\{B\}$ is incorrectly mapping to $\{D\}$). Instances that failed to reproduce (red ceils) are due to the inaccuracies on the last index as our method depends on the last index to end the search (i.e., in Figure 9(c), we search the execution trace on $\{E\}$).

5 USEFULNESS EVALUATION

We conduct a user study to evaluate the usefulness of the generated execution trace for replaying visual bug recording into real-world development environments. We recruit 8 participants including 6 graduate students (4 Master, 2 Ph.D) and 2 software developers to participate in the experiment. All students have at least one-year experience in developing Android apps and have worked on at least one Android apps project as interns in the company. Two software developers are more professional and work in a large company (Alibaba) about Android development.

Procedure: We first give them an introduction to our study and also a real example to try. Each participant is then asked to reproduce the same set of 10 randomly selected visual bug recordings from GitHub which are of diverse difficulty ranging from 6 to 11 steps until triggering bugs. Detailed experiment dataset can be seen in our online appendix[5]. The study involves two groups of four participants: the experimental group P_1, P_2, P_3, P_4 who gets help with the generated execution trace by our tool, and the control group P_5, P_6, P_7, P_8 who starts from scratch. Each pair of participants $\langle P_x , P_{x+4}\rangle$ have comparable development experience, so that the experimental group has similar capability to the control group in total. Note that we do not ask participants to finish half of the tasks with our tool while the other half without assisting tool to avoid potential tool bias. Our tool takes about 138.08s on average to generate execution trace for the average 10.59s bug recording as seen in the Table 6. We only record the time used to reproduce the visual bug recordings in Android, as the execution trace generation can be finished offline, especially for long recordings which require much processing time. Since our approach is fully automated, our model can automatically deal with the bug video recording immediately once uploaded. Participants have up to 10 minutes for each bug replay.

Results: Table 7 shows the experiment result. Although most participants from both experimental and control groups can successfully finish the bug replay on time, the experiment group reproduces the visual bug recording much faster than that of the control group (with an average of 171.4 seconds versus 65.0 seconds). In fact, the average time of the control group is underestimated, because three bugs fail to be reproduced within 10 minutes, which means that participants may need more time. In contrast, all participants in the experiment group finish all the tasks within 2 minutes. To understand the significance of the differences between the two groups, we carry out the Mann-Whitney U test [54] (specifically designed for small samples) on the replaying time. The testing result suggests that our tool can significantly help the experimental group reproduce bug recordings more efficiently ($p - value < 0.01$).

We summarise two reasons why it takes the control group more time to finish the reproduction than the experiment group. First, some visual recording is quite complicated which requires participants in the control group to watch the visual recordings several times for following procedures. The GUI transitions within the recording may also be too fast to follow, so developers have to replay it. Second, it is hard to determine the trigger from one GUI to the next one. As illustrated in Section 2.3, only 25% of videos are recorded with the touch indicator, resulting in developers' guess of the action for triggering the next GUI. That trial and error makes the bug replay process tedious and time-consuming. It is especially severe for junior developers who are not familiar with the app code.

6 THREATS TO VALIDITY

We had discussed the limitations of our approach at the end of each subsection of the evaluation in Section 4, such as errors due to slow rendering in keyframe location (Section 4.1), dynamic loading content in GUI mapping (Section 4.2), etc. In this section, we further discuss the threats to validity of our approach.

Internal Validity. In our automated experiments evaluating GIFdroid, threats to internal validity may arise from human annotators and artificial recordings. To help mitigate this threat, we

[5]https://sites.google.com/view/gifdroid

Table 5: Detailed results for execution trace generations. Green cells indicate fully reproduced recordings, orange cells > 50% reproduced, and red cells < 50%.

AppName	Rep. Trace		AppName	Rep. Trace		AppName	Rep. Trace		AppName	Rep. Trace	
Token	8/8	5/5	TimeTracker	10/10	8/8	TrackerControl	23/24	12/12	YoloSec	9/9	8/10
DeadHash	25/25	18/18	GNUCash	9/9	7/7	aFreeRDP	11/11	10/10	AntennaPod	7/7	8/9
ProtonVPN	7/7	9/9	FastNFitness	8/8	8/10	JioFi	6/6	8/8	WiFiAnalyzer	9/9	5/5
PSLab	8/9	6/6	DroidWeight	6/6	7/7	openScale	18/18	3/18	KeePassDX	6/6	7/7
Trigger	7/7	12/12	ADBungFu	2/8	21/21	EteSyncNotes	11/11	12/12	PortAuthority	8/8	12/12
ATime	7/7	8/8	AdAway	11/11	19/19	StinglePhoto	7/7	6/6	InviZible	6/6	10/10

Table 6: The execution time of `GIFdroid` for real-world bug recordings

AppName	Gif	`GIFdroid` Execution			
	Duration (sec)	Keyframe Location (sec)	GUI Mapping (sec)	Trace Generation (sec)	Total (sec)
AnkiDroid-1	16.7	28.05	174.96	0.58	203.59
AnkiDroid-2	9.5	13.95	84.80	1.12	99.87
KISS-1	13.8	16.74	148.51	0.13	165.38
NeuroLab-1	9.6	14.76	71.05	0.01	85.82
NeuroLab-2	4.5	4.76	77.32	0.13	82.21
BeHe-1	11.8	7.10	168.77	0.11	175.98
BeHe-2	6.0	3.73	64.80	0.60	69.13
AndrOBD-1	7.9	10.00	115.56	0.41	125.97
PDFConverter-1	17.4	36.22	191.03	2.32	229.57
PDFConverter-2	8.7	9.99	131.51	1.84	143.34
Average	10.59	14.53	122.83	0.72	138.08

Table 7: Performance comparison between the experimental and control group. * denotes $p < 0.01$.

AppName	Control Group		Experimenal Group	
	Success	Time (sec)	Success	Time (sec)
AnkiDroid-1	2/4	473	4/4	102
AnkiDroid-2	3/4	334	4/4	95
KISS-1	4/4	121	4/4	82
NeuroLab-1	4/4	90	4/4	73
NeuroLab-2	4/4	366	4/4	62
BeHe-1	4/4	77	4/4	55
BeHe-2	4/4	94	4/4	51
AndrOBD-1	4/4	63	4/4	56
PDFConverter-1	4/4	71	4/4	47
PDFConverter-2	4/4	25	4/4	24
Average	3.7/4	171.4	4/4	65.0*

recruited two paid annotators from online posting who have experience in video annotation. To mitigate any potential subjectivity or errors, we asked them to annotate independently without any discussion, and then met and sanity corrected the discrepancies. Another potential threat concerns the selection of optimal execution trace. To mitigate this threat, we chose the shortest candidate sequence as the optimal execution trace to help developers reproduce the bug with the least amount of time/steps.

External Validity. The main threat to external validity arises from the potential bias in the selection of experimental apps used in our automated evaluation. To help mitigate this threat, we utilized the apps used in previous studies [23, 26, 55], which have undergone several filtering and quality control mechanisms to ensure diversity. One more potential external threat concerns the generalizability on the manual creation of visual recordings for validating the performance of our approach. For example, there are many different types of devices with different screens especially in Android which may result in different GUI rendering. To mitigate this threat, we took care to generate the recordings as general as possible, including different creation tools, varied resolutions, diverse length and differed recording speed.

7 RELATED WORK

A growing body of tools has been dedicated to assisting in recording and replaying bugs in mobile and web apps. We introduce related works of bug replay based on different types of information including the app running information, textual description, and visual recording in this section.

7.1 Bug Record and Replay from App Running Information

Nurmuradov et al. [57] introduced a record and replay tool for Android applications that captures user interactions by displaying the device screen in a web browser. It used event data captured during the recording process to generate a heatmap that facilitates developers understanding of how users are interacting with an application. Additional tools including ECHO [69], Reran [39], Barista [42], and Android Bot Maker [2] are program-analysis related applications for the developer to record and replay interactions. However, they required the installation of underlying frameworks such as replaykit [4], troyd [40], or instrumenting apps which is too heavy

for end users. In contrast to these approaches, our GIFdroid is rather light-weight which just requires the input recording and UTG which can be generated by existing tools.

7.2 Bug Replay from Textual Information

To lower the usage barrier of record and replay that requires frameworks, many of the studies [22, 37, 41, 55, 74] facilitate the bug replay base on the natural language descriptions in bug report which contains immediately actionable knowledge (e.g., reliable reproduction steps), stack traces, test cases, etc. For example, ReCDroid [84] leveraged the lexical knowledge in the bug reports and proposed a natural language processing (NLP) method to automatically reproduce crashes. However, it highly depended on the description writing in the bug report including formatting, word usage, granularity [24, 36, 43] which limit its deployment in the real world. Many works that assist developers in writing bug reports [51, 53, 55, 56] still cannot directly benefit it. Different from textual bug reports, visual recordings contain more bug details and can be easily created by non-technical users. Therefore, our work focuses on the automated bug replay from these visual bug reports.

7.3 Bug Replay from Visual Information

In software engineering, there are many works on visual artifacts including recording, tutorials, bug reports [27, 29, 32, 38, 61, 78, 82] with some of them specifically for usability and accessibility testing [30, 49, 76, 81]. In detail, Bao et al. [20] focused on the extraction of user interactions to facilitate behavioral analysis of developers during programming tasks, such as code editing, text selection, and window scrolling. Krieter et al. [44] analyzed every single frame of a video to generate log files that describe what events are happening at the app level (e.g., the "k" key is pressed). Different from these works in extracting developers' behavior, our work is concerned with the automated bug replay from the recording in the bug report.

Bernal et al [23] proposed a tool named V2S which leverages deep learning techniques to detect and classify user actions (e.g., a touch indicator that gives visual feedback when a user presses on the device screen) captured in a video recording, and translate it into a replayable test script. Although the target of this work is the same as ours, there are two major differences between them. First, V2S requires high-resolution recording with touch indicators which is hard to get in real-world bug reports according to our analysis (less than 25.8%) in Section 2.2. Second, it requires complete recording from the app launch to the bug occurrence while most recordings (93.2%) in the real world do not start from the app launch, but just 2-7 steps before the bug page. In contrast, our approach design is fully driven by the practical data, hence it does not have those requirements. The user study of real-world bug replay also confirms the usefulness and generality of our approach.

8 CONCLUSION

The visual bug recording is trending in bug reports due to its easy creation and rich information. To help developers automatically reproduce those bugs, we propose GIFdroid, an image-processing approach to covert the recording to executable trace to trigger the bug in the Android app. Our automated evaluation shows that our GIFdroid can successfully reproduce 82% (50/61) visual recordings from 31 Android apps. The user study on replaying 10 real-world visual bug recordings confirms the usefulness of our GIFdroid in boosting developers' productivity.

In the future, we will keep improving our method for better performance in term of keyframe extraction, GUI mapping, and trace generation. For example, the traditional image processing methods may not robust to minor GUI changes such as configuration and parameter changes. We will further improve our approach to locate more fine-grained parameter information. To make GIFdroid more usable, we will also take the human factor into the consideration. As the automated approach may not be perfect, we will further explore how human can collaborate with the machine for replaying the bugs in the visual bug report. While GIFdroid is fully automated, can run in the background, the execution overhead is not ideal. In the future, we will improve the efficiency of our approach, for example, accelerating by more advanced hardware and algorithm.

REFERENCES

[1] 2021. Android Developer: Playback capture. https://developer.android.com/guide/topics/media/playback-capture.
[2] 2021. Bot Maker for Android - Apps on Google Play. https://play.google.com/store/apps/details?id=com.frapeti.androidbotmaker.
[3] 2021. BugClipper. https://bugclipper.com/.
[4] 2021. Command line tools for recording, replaying and mirroring touchscreen events for Android. https://github.com/appetizerio/replaykit.
[5] 2021. F-Droid. https://f-droid.org/en/packages/.
[6] 2021. Firebase - Build and Run Successful Apps. https://firebase.google.com/.
[7] 2021. GIPHY: Search All the GIFs, Make Your Own Animated GIF. https://giphy.com/.
[8] 2021. GitHub. https://github.com/.
[9] 2021. Python and OpenCV-based scene cut/transition detection program & library. https://github.com/Breakthrough/PySceneDetect.
[10] 2021. Record the screen on your iPhone, iPad, or iPod touch. https://support.apple.com/en-us/HT207935.
[11] 2021. Sketch — The digital design toolkit. https://www.sketch.com/.
[12] 2021. Take a screenshot or record your screen on your Android device. https://support.google.com/android/answer/9075928?hl=en.
[13] 2021. TestFairy. https://www.testfairy.com/.
[14] 2021. VirtualDub.org. https://www.virtualdub.org/index.
[15] 2021. YouTube. https://www.youtube.com/.
[16] Mohammad A Alsmirat, Fatimah Al-Alem, Mahmoud Al-Ayyoub, Yaser Jararweh, and Brij Gupta. 2019. Impact of digital fingerprint image quality on the fingerprint recognition accuracy. *Multimedia Tools and Applications* 78, 3 (2019), 3649–3688.
[17] John Anvik, Lyndon Hiew, and Gail C Murphy. 2005. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. 35–39.
[18] Jorge Aranda and Gina Venolia. 2009. The secret life of bugs: Going past the errors and omissions in software repositories. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 298–308.
[19] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 641–660.
[20] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, Xin Xia, and Bo Zhou. 2017. Extracting and analyzing time-series HCI data from screen-captured task videos. *Empirical Software Engineering* 22, 1 (2017), 134–174.
[21] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded up robust features. In *European conference on computer vision*. Springer, 404–417.
[22] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight recording to reproduce field failures. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 362–371.
[23] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 309–321.
[24] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software*

engineering. 308–318.
[25] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories*. 27–30.
[26] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 86–96.
[27] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.
[28] Hu Chen, Mingzhe Sun, and Eckehard Steinbach. 2009. Compression of Bayer-pattern video sequences using adjusted chroma subsampling. *IEEE transactions on circuits and systems for video technology* 19, 12 (2009), 1891–1896.
[29] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI design search through image autoencoder. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–31.
[30] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 322–334.
[31] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 596–607.
[32] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports. *arXiv preprint arXiv:2101.09194* (2021).
[33] Per-Erik Danielsson. 1980. Euclidean distance mapping. *Computer Graphics and image processing* 14, 3 (1980), 227–248.
[34] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 845–854.
[35] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction mining mobile apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. 767–776.
[36] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 62–71.
[37] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.
[38] Christian Frisson, Sylvain Malacria, Gilles Bailly, and Thierry Dutoit. 2016. InspectorWidget: A System to Analyze Users Behaviors in Their Applications. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 1548–1554.
[39] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 72–81.
[40] Jinseong Jeon and Jeffrey S Foster. 2012. *Troyd: Integration testing for android*. Technical Report.
[41] Fitsum Meshesha Kifetew, Wei Jin, Roberto Tiella, Alessandro Orso, and Paolo Tonella. 2014. Reproducing field failures for programs with complex grammar-based input. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 163–172.
[42] Andrew J Ko and Brad A Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 387–396.
[43] A Gunes Koru and Jeff Tian. 2004. Defect handling in medium and large open source projects. *IEEE software* 21, 4 (2004), 54–61.
[44] Philipp Krieter and Andreas Breiter. 2018. Analyzing mobile application usage: generating log files from mobile screen recordings. In *Proceedings of the 20th international conference on human-computer interaction with mobile devices and services*. 1–10.
[45] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for android: Are we there yet in industrial cases?. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 854–859.
[46] Daeho Lee and Sungsoo Lim. 2016. Improved structural similarity metric for the visible quality measurement of images. *Journal of Electronic Imaging* 25, 6 (2016), 063015.
[47] Toby Jia-Jun Li, Lindsay Popowski, Tom M Mitchell, and Brad A Myers. 2021. Screen2Vec: Semantic Embedding of GUI Screens and GUI Components. *arXiv preprint arXiv:2101.11103* (2021).
[48] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.
[49] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting ui display issues via visual understanding. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 398–409.
[50] Margaret Livingstone and David H Hubel. 2002. *Vision and art: The biology of seeing*. Vol. 2. Harry N. Abrams New York.
[51] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. 2015. Modelling the 'hurried'bug report reading process to summarize bug reports. *Empirical Software Engineering* 20, 2 (2015), 516–548.
[52] David G Lowe. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision* 60, 2 (2004), 91–110.
[53] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. 2012. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
[54] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.
[55] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing bug reports for android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 673–686.
[56] Laura Moreno and Andrian Marcus. 2017. Automatic software summarization: the state of the art. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 511–512.
[57] Dmitry Nurmuradov and Renee Bryce. 2017. Caret-HM: recording and replaying Android user sessions with heat map generation using UI state clustering. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 400–403.
[58] Paul Over, George Awad, Martial Michel, Jonathan Fiscus, Greg Sanders, Barbara Shaw, Wessel Kraaij, Alan F Smeaton, and Georges Quéot. 2013. Trecvid 2012-an overview of the goals, tasks, data, evaluation mechanisms and metrics. (2013).
[59] Maciej Pesko, Adam Svystun, Paweł Andruszkiewicz, Przemysław Rokita, and Tomasz Trzciński. 2019. Comixify: Transform Video Into Comics. *Fundamenta Informaticae* 168, 2-4 (2019), 311–333.
[60] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002).
[61] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Too long; didn't watch! extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering*. 261–272.
[62] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*. 571–582.
[63] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *2011 International conference on computer vision*. Ieee, 2564–2571.
[64] Yair Shemer, Daniel Rotman, and Nahum Shimkin. 2019. ILS-SUMM: Iterated Local Search for Unsupervised Video Summarization. *arXiv preprint arXiv:1912.03650* (2019).
[65] Yang Song and Oscar Chaparro. 2020. BEE: a tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1551–1555.
[66] Yale Song, Miriam Redi, Jordi Vallmitjana, and Alejandro Jaimes. 2016. To click or not to click: Automatic selection of beautiful thumbnails from videos. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 659–668.
[67] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[68] Ramadass Sudhir and Lt Dr S Santhosh Baboo. 2011. An efficient CBIR technique with YUV color space and texture features. *Computer Engineering and Intelligent Systems* 2, 6 (2011), 78–85.
[69] Yulei Sui, Yifei Zhang, Wei Zheng, Manqing Zhang, and Jingling Xue. 2019. Event trace reduction for effective bug replay of Android apps via differential GUI state analysis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1095–1099.
[70] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. 2012. Automating test automation. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 881–891.

[71] Shiqi Wang, Abdul Rehman, Zhou Wang, Siwei Ma, and Wen Gao. 2011. SSIM-motivated rate-distortion optimization for video coding. *IEEE Transactions on Circuits and Systems for Video Technology* 22, 4 (2011), 516–529.
[72] Xiang-Yang Wang, Jun-Feng Wu, and Hong-Ying Yang. 2010. Robust image retrieval based on color histogram of local feature regions. *Multimedia Tools and Applications* 49, 2 (2010), 323–345.
[73] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.
[74] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Generating reproducible and replayable bug reports from android application crashes. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 48–59.
[75] Krzysztof Wołk and Krzysztof Marasek. 2014. A sentence meaning based alignment method for parallel text corpora preparation. In *New Perspectives in Information Systems and Technologies, Volume 1*. Springer, 229–237.
[76] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. 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, 761–772.
[77] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.
[78] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and Image Recognition Driving Cross-Platform Automated Mobile Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1561–1571.
[79] Semir Zeki. 1993. *A vision of the brain*. Blackwell scientific publications.
[80] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: Vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 350–361.
[81] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: vision-based linting of GUI animation effects against design-don't guidelines. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1286–1297.
[82] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 748–760.
[83] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically Extracting Bug Reproducing Steps from Android Bug Reports. In *International Conference on Software and Systems Reuse*. Springer, 100–111.
[84] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139.