Extracting Replayable Interactions from Videos of Mobile App Usage

Jieshan Chen*
Australian National University
Canberra, Australia
jieshan.chen@anu.edu.au

Amanda Swearngin
Apple
Seattle, WA, USA
aswearngin@apple.com

Jason Wu
Apple
Seattle, WA, USA
jason_wu2@apple.com

Titus Barik
Apple
Seattle, WA, USA
tbarik@apple.com

Jeffrey Nichols
Apple
Seattle, WA, USA
jwnichols@apple.com

Xiaoyi Zhang
Apple
Seattle, WA, USA
xiaoyiz@apple.com

ABSTRACT

Screen recordings of mobile apps are a popular and readily available way for users to share how they interact with apps, such as in online tutorial videos, user reviews, or as attachments in bug reports. Unfortunately, both people and systems can find it difficult to reproduce touch-driven interactions from video pixel data alone. In this paper, we introduce an approach to extract and replay user interactions in videos of mobile apps, using only pixel information in video frames. To identify interactions, we apply heuristic-based image processing and convolutional deep learning to segment screen recordings, classify the interaction in each segment, and locate the interaction point. To replay interactions on another device, we match elements on app screens using UI element detection. We evaluate the feasibility of our pixel-based approach using two datasets: the Rico mobile app dataset and a new dataset of 64 apps with both iOS and Android versions. On these datasets, we evaluate the performance of our video segmentation, interaction classification, and interaction localization methods with insights. Our approach can successfully replay a majority of interactions (iOS−84.1%, Android−78.4%) on different devices, which is a step towards supporting a variety of scenarios, including automatically annotating interactions in existing videos, automated UI testing, and creating interactive app tutorials.

KEYWORDS

video record and replay, video segmentation, video classification, action localization, mobile applications

*Work done at Apple.

1 INTRODUCTION

Videos of mobile app usage have become commonplace on the internet. For example, Tech vloggers make app tutorials to educate new users or share the best practices of app features. People record app screens step-by-step to guide their parents who are not familiar with an app, and app testers create bug reports with rich context in video. In particular, screen recordings—that is, videos that record on-screen content but not the user's hands—are easy to produce and share using built-in smartphone facilities.

In order to effectively use these screen recordings however, the viewer has to first understand the interactions performed in the video and then manually repeat them in the order shown. This process can be time-consuming and error-prone [2], especially when the sequence of necessary interactions is long or the recording is played quickly. For example, a viewer may have to pause the recording after each interaction, or even replay it multiple times before they are able to replicate the interaction on their own device. In addition, the target UI element for an interaction may be difficult to locate in complex app screens, in the presence of different display preferences—such as large fonts or dark mode—or when device or app versions change.

What if instead of asking a user to do all of the work of interpreting a screen recording, a machine could do it instead? Such a system might identify the type of interactions that were performed and the target UI elements that these interactions were performed on, ideally from the recording alone. Work has been attempted in this area before, but a limitation of existing systems is that they require a special recording apparatus [16, 29], visual indicators added during recording [2, 21] or additional metadata such as UI transition graph [9] in order to capture the interactions demonstrated in the video.

In this paper, we propose a system that automatically extracts user interactions from ordinary screen recordings, without requiring additional settings, specially-instrumented recording tools, or source code access. Our system performs three phrases to extract the interactions: 1) video segmentation, 2) interaction classification, and 3) interaction localization. Our system first segments the start and the end of each interaction. Then, it runs heuristics to choose between six common interaction types. Finally, our system uses a 3D convolutional encoder to learn the semantics of the animation in the UI, a 2D convolutional encoder and decoder to capture the connections between consecutive UI states, and another decoder to

infer the interaction probability heatmap and output the location of the interaction. Based on UI detection results from screenshot pixels, we can find the target UI element and its content. We also explore methods to replay the interactions that we extract from videos on another device. Our replay prototype runs UI detection on each app screen, locates the best matched UI element for each recorded interaction and performs the interaction on each UI element in turn. Table 1 summarizes the key differences between our method and existing techniques.

We evaluated our system on the Rico dataset [7] (created 4 years ago), and a smaller app usage dataset (64 top-downloaded apps, each has iOS and Android versions) that we collected and annotated recently. For video segmentation, our system achieves 84.7% recall on iOS, and slightly worse recall on Android (72.0%). For interaction classification, our system achieves comparable accuracy on both platforms (iOS−87.6%, Android−89.3%). For interaction localization, our system achieves the best accuracy on Rico (69.1%), as the model is trained on this dataset. Although app UI design changes have occurred since Rico, our model still works on recent Android with lower accuracy (56.2%), and 41.4% accuracy on recent iOS apps. For interaction replay, we found that the majority of interactions (iOS−84.1%, Android−78.4%) can be replayed on different devices.

The contributions of this paper are as follows:

- We present a pixel-based approach to automatically extract and replay interactions from ordinary screen recordings without requiring additional settings, specialized recording instrumentation, or access to source code.
- We implement a prototype system that instantiates our approach. The results of our evaluation show reasonable accuracy in video segmentation, interaction classification, and interaction localization. Our system successfully replays a majority of interactions on different devices. These results demonstrate the feasibility of our pixel-based approach to extracting replayable interactions.

2 RELATED WORK

We discuss the related work across two areas: 1) identifying interactions on user interfaces, and 2) replaying user interactions.

Table 1: Differences in input, additional data requirements, and support for cross-device replay between existing techniques and our proposed system.

	Input	Additional Requirement	Cross-Device Replay
APPINITE[16]	User demonstration	View hierarchy, Interaction point & type	No
V2S[2]	Video	Add touch indicator to the video	No
RoScript[21]	Test script or Video	Include fingertips	No
LIRAT[29]	Test script or User demonstration	Interaction point & type	Yes
GifDroid[9]	Video and UI Transition Graph	None	No
Our system	**Video**	**None**	**Yes**

2.1 Identifying Interactions on User Interfaces

Identifying user interactions is an important task on various platforms, including mobile [2, 9, 16, 21, 29], desktop [19, 32], web [1, 24], and even the physical world [10]. The extracted interactions can empower many applications, including task automation [16], bug reporting [2], automated app testing [21, 24, 29], and guidance to use appliances [10].

Previous research has applied various methods to identify user interactions for the purposes of replaying them on the same or another device. LIRAT [29] obtains the interaction location and type by using a debugging tool to access low-level system events. The device must also have a connection to a computer that runs the debugging tool. APPINITE [16] adds a layer of interaction proxies [31] on top of the current running app. An interaction proxy layer captures the users' taps and passes them to the underlying app. This method requires installing an additional background service and obtaining Accessibility permissions, and may not work on all platforms. V2S [2] requires users to access Android developer settings in order to show a touch indicator at each *tap* event. With this known visual indicator, V2S presents an object detection model to locate the touch indicator and infer the interaction location. This method adds extra work to app video creators, and not all video creators would like to show a developer-mode touch indicator in videos. RoScript [21] instead requires video creators to use an external camera to record the phone screen and finger movement. It leverages computer vision techniques to recognize a finger and its relative location to the app UI, and the system requires users to move their fingers outside the phone screen between each interaction for segmentation. This method also requires an additional camera and a stable setup of phone and camera. While GifDroid [9] does not require complex setup of the input video, it additionally relies on the UI transition graph to assist interaction identification. However, constructing UI transition graph is not be a trivial graph and requires many efforts.

The methods above require settings or recording tools that are specific to a platform (e.g., Android) [16, 29], add extra work to app video creators [2, 21], and will not work on existing app usage videos [2, 16, 21, 29]. We believe our pixel-based approach can be a more generalizable way to collect interaction traces from videos.

2.2 Replaying User Interactions

After extracting user interactions, the key challenge of replaying is to find where to interact on the replay device. Some work repeats the (x, y) coordinate from the recording [2, 11], while some applications [21, 29] find matching UI elements on replaying devices so that the replay will be more robust to dynamic content and device change.

To find a matching UI element sometimes requires access to a view hierarchy. For example, APPINITE [16] tries to match UI metadata from the view hierarchy (e.g., parent-child relationship, text, UI Class) so that it can still locate the target UI element even when the target UI element moves to a different location due to screen content change. However, the view hierarchy is not always available, and the view hierarchy can be incomplete or misleading.

To avoid these limitations, some work leverage computer vision techniques to match targeted UI elements. For example, LIRAT [29]

compares image features to match UI elements between recording and replaying screens, and extracted layout hierarchy from pixels to improve matching. Similarly, our method also leverages video pixels to match target UI elements, but we use object detection models that have better performance.

3 SYSTEM

Essentially, our system extracts interactions from pixels in video frames, and uses this information to replay the interactions on another device. As shown in Figure 1, extraction is performed through three phases: 1) video segmentation (Section 3.1), interaction classification (Section 3.2), and interaction localization (Section 3.3). Our system then applies a set of strategies in the interaction replay phase (Section 3.4). The rest of this section describes the system phases in detail.

3.1 Phase 1—Video Segmentation

Video segmentation splits the frames of the input screen recording into a sequence of representative frames that maximally differentiates the video, or *keyframes* [33].

To illustrate how this works, consider the frames of the screen recording shown in Figure 2. We start by computing the histogram of oriented gradient (HOG) [6] feature descriptor for each frame. As the name suggests, the HOG descriptor is a simplified representation of the screen in terms of its structure or shape through gradient and orientation. We use this HOG descriptor to calculate similarity between consequence frames using a structural similarity (SSIM) measure [28]. Intuitively, a sequence of similar frames represents a stable interval—with the middle of this stable interval being the keyframe, represented with an arrow in Figure 2.

Next, we run a spike detection algorithm using empirical parameters we derived from a number of app usage videos (Section 4): 1) the spike should be larger than $(\text{Similarities}_{max} - \text{Similarities}_{min})/15$ [1], to be resilient to partial content changes that are not caused by user interactions, and 2) the stable intervals should contain at least four frames, to mitigate against interactions from transient UI changes. The inverted spikes in Figure 2 indicate interaction clip points that segment the keyframes.

3.2 Phase 2—Interaction Classification

This phase identifies six interactions—*type*, *swipe left*, *swipe right*, *swipe up*, *swipe down*, and *tap* [17]—through the following heuristics:

Type interactions are always associated with a virtual keyboard. We inspect the OCR results on a screen to determine if they contain text corresponding to the rows of a keyboard—that is, 3 rows of QWERTY keyboard, and 4 rows of number pad (Figure 3(a)). For entered text, we compare the OCR results in the first and last frames of a *type* interaction. To illustrate, in Figure 3(b)'s right-most screen, the placeholder inside the top search bar changes as text is entered, with suggestions appearing below. Among all changed or added OCR text results in the last frame, we pick the top-most one (smallest y) as the entered text.

Swipe (left, right, up, down) interactions shift several UI elements within a scrollable area, while the top title bar and bottom

---

tab bar are often unchanged (see Figure 6 and Figure 7 in the Appendix). Consequently, we compare the OCR results between any two consecutive frames and calculate the movement between each pair of text strings. If multiple ($N >= 3$, an empirical parameter) text strings move in the same horizontal or vertical direction within a threshold distance, our system classifies the interaction as a *swipe*. We call the text strings with shared movement a *text collection*. Note that sometimes "snackbar" elements may briefly appear at the bottom of the screen with messages about app processes, so we set $N >= 3$ to avoid confusing this UI behavior with a swipe.

Capturing the semantics of *swipe* interaction requires three properties: the direction, the distance, and initiation point. The Swipe direction is determined trivially through the movement coordinates. To calculate the *swipe* distance, we use the median distance of movement between two consecutive frames, and then sum all median distances between any two consecutive frames in the interaction clip. The *swipe* initiation point is the either first or last text by x or y position, for horizontal or vertical *swipe*, respectively.

Tap interactions may lead to a new UI state, pop-up keyboard, or cause few or no movement of shared elements. If an interaction clip is not classified as either a *type* or *swipe*, we classify it as a a *tap* interaction. From the Rico dataset [7], we found that the majority of interactions in mobile apps are *tap* interactions (91.7%), and that *type* and *swipe* interactions often cause changes on text elements—for example, through creating or moving text. Informed by these findings, treating *tap* as the fall-through interaction has reasonable justification.

For a *tap* interaction, we must also identify the *tap* location. This is described in the next section.

3.3 Phase 3—Interaction Localization for Tap Interactions

For *tap* interactions, interaction localization is needed to identify the location of the *tap*. In some cases, the start and end UI state will share an interaction component. For this situation, we can use heuristic-based localization (Section 3.3.1) to identify the *tap* interaction location.

When heuristic-based location fails, we can rely on visual feedback cues provided by the app when the users *tap* a location. In this situation, we leverage the animation effect and the connections between the two consecutive UI states to train an interaction localization model to locate the interaction point (Section 3.3.2).

3.3.1 Heuristic-based Localization. When the title of the new UI state is same as the label of one items in the content area, it is likely that this is the item the user *tap*ped. In Figure 4(b), when users *tap* on "History", the title of the new UI state also becomes "History". Because this is a high-accurate heuristic, we first detect the existence of this pattern to locate the interacted element (Section 3.3.1).

To detect the title, we first run OCR on the first and last frame of each interaction clip to obtain all texts. We then find the top title in the second frame, and check if there is an element with the same text in the main content—excluding the top bar and the bottom app bar. If so, we output the position of this element as current *tap* interaction point.

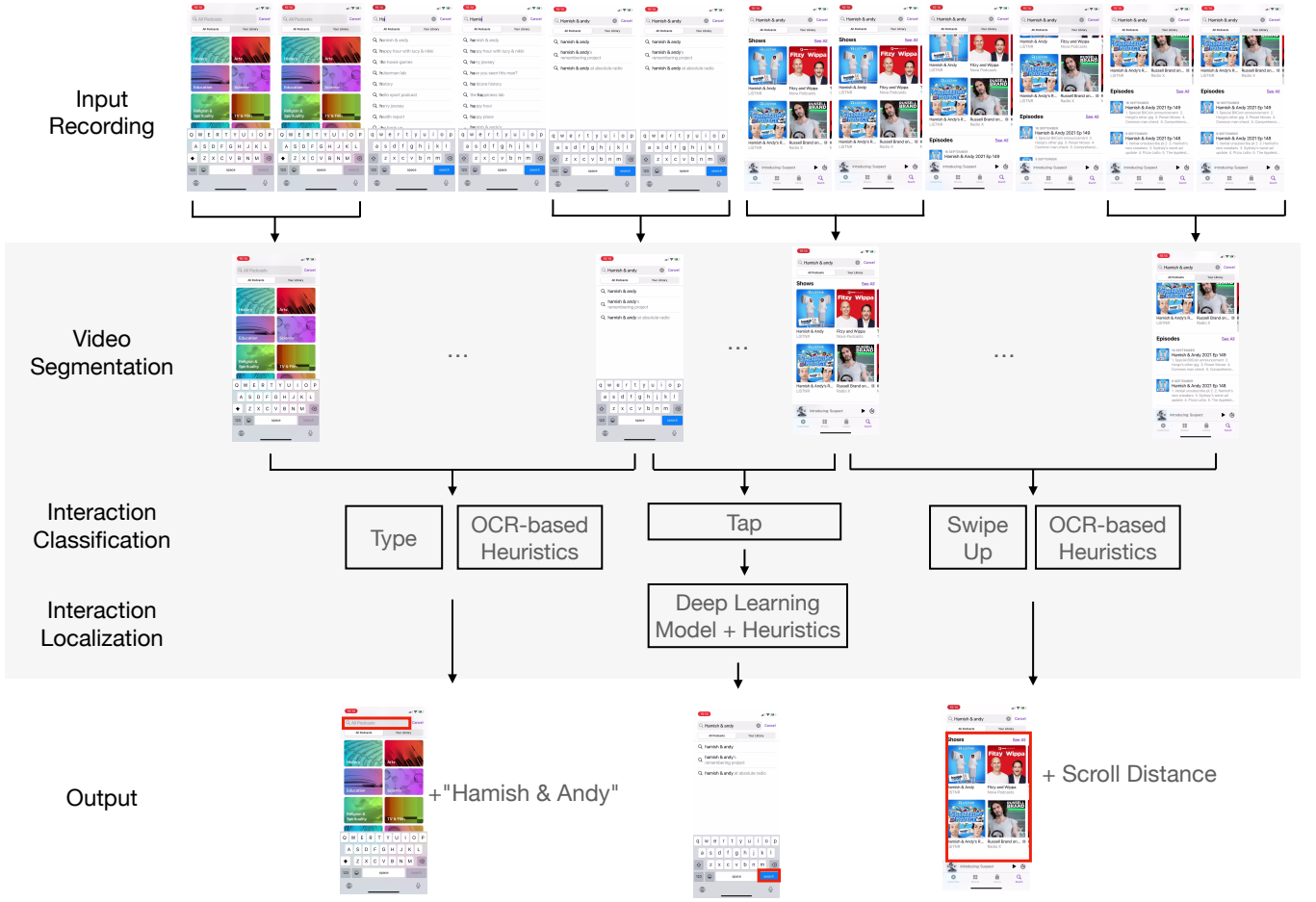[1]15 is an empirically set number from experiments.

Figure 1: Flowchart of our system: First, it segments the frames from an input screen recording into a sequence of representative frames (i.e., keyframes). Second, it applies heuristics to classify the interactions into six common types. For *type* and *swipe* interactions, it relies on the OCR heuristics to locate the interaction point and the corresponding information (typed text for *type* interactions, scroll distance for *swipe* interactions). For *tap* interactions, it applies both heuristics and a deep learning based interaction localization model to determine the interaction point. The overall output is an interaction trace that can be replayed on another device.

3.3.2 Localization Model. From our observations of the Rico animation data and our collected app usage videos, we identified three common visual cues that we can leverage to locate *tap* interactions: 1) ripple effect—a radial action in the form of a visual ripple expanding outward from the user's touch, 2) expand effect—which scales up and cross-fades a UI elements, and 3) changes in the text or background colors. In addition, we noticed that in some cases, the start and end UI state shares the interacted element. For example, in Figure 4(a), when users *tap* "Hotels That Are Homes for the Harvest", the new UI state contains the same text. However, although in this case, the shared element is the interacted element, it may fail in other situation. We rely on our models to learn the difference between normal shared elements and shared interacted elements.

We trained a deep-learning based interaction localization model to locate the *tap* point. Inspired by the success of human pose recognition [20], object detection [15, 18], and video classification models [25], we designed a model to predict a heatmap of possible *tap* points by learning the semantics of the animation effects.

As shown in Figure 5, our model primarily consists of three blocks: a 2D block, a 3D block and a decoder. The 2D block is based on 2D convolutional layers [20] and aims to find the connections between two consecutive UI states, while the 3D block—based on 3D convolutional layers [25]—captures the temporal relationship among frames—that is, the animation effect across these frames—in each interaction. A final decoder then fuses features extracted from the 2D and the 3D block to infer the interaction heatmap. We added a shortcut module following the U-Net model [22] to help the model to retrieve the coarse features from the shallow layer in the encoder part to refine the extracted high-level abstract features and help dense per-pixel heatmap prediction. Concretely, given 8 frames extracted from one interaction clip as the input of our model
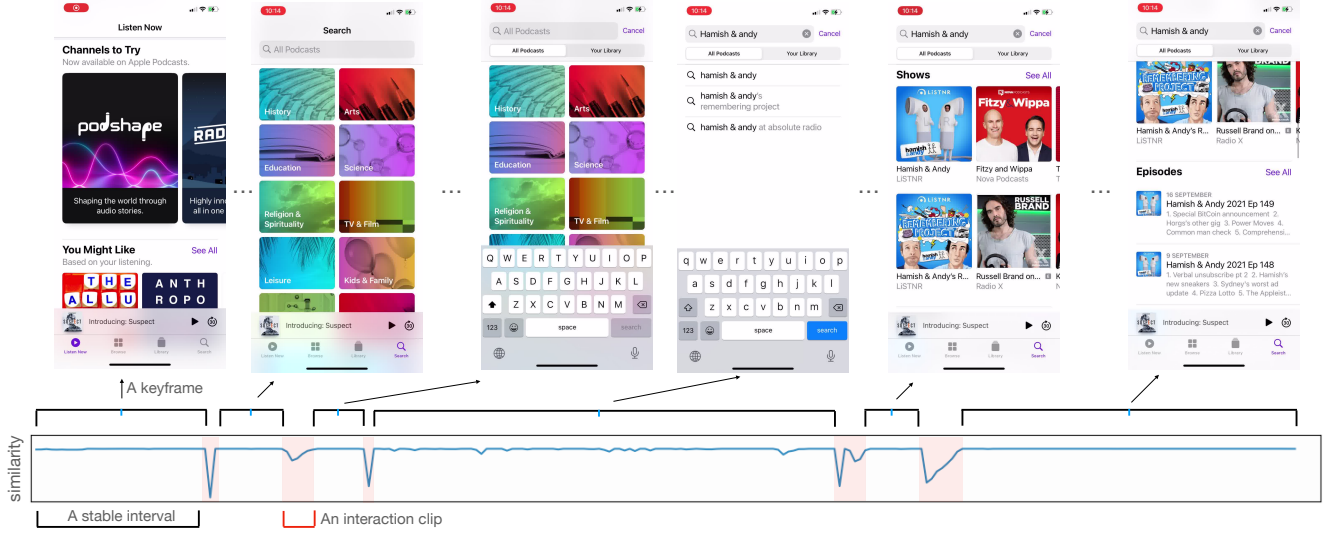
Figure 2: Visualization of image similarities between consecutive frames. The spikes indicate segments in the video where user interactions were performed, which we use to segment the keyframes. In this figure, we detected six stable intervals, and for each interval, we take the middle frame as the extracted keyframe. The users *tap* on the bottom-right "search" icon, *tap* the input field on the top, *type* text, *tap* the bottom-right "search" icon, and *swipe* up to see more content.

(a) Keyboard Patterns

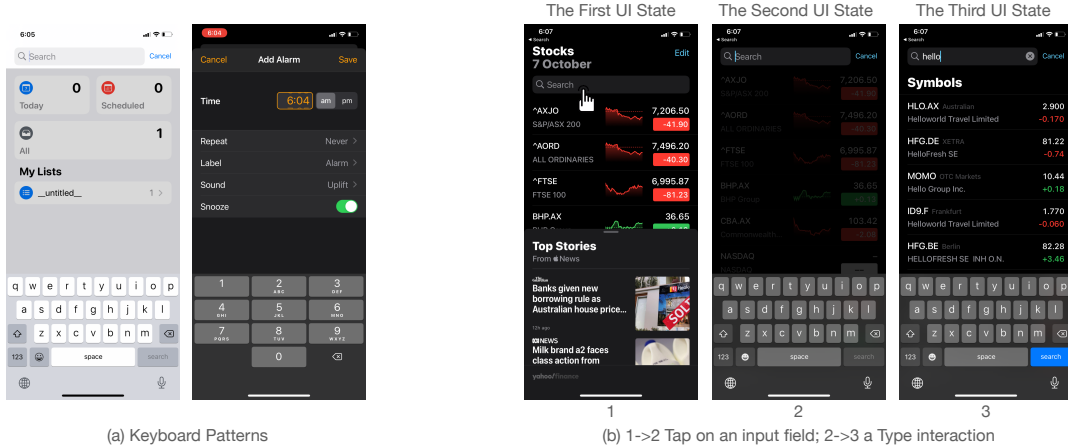(b) 1->2 Tap on an input field; 2->3 a Type interaction

Figure 3: Examples of patterns that our interaction classification heuristics examine to classify a *type* interaction, including (a) Keyboard patterns that can be detected from OCR results. (b) When users *tap* on an input field, the title of the UI will change instantly; when users perform a *type* interaction, the title will have a steady change or remain the same.

(Section 4.4), we take the first and the last frames as the input to 2D block to learn the connections between the two consecutive UI states. In parallel, we feed all frames into our 3D block to encode the animation effect: the extracted features from 2D block and 3D block will then be concatenated, and the combined feature will be fed into the decoder to predict the interaction heatmap, which is the output of our localization model. We take the point with highest probability in the predicted heatmap as the output interaction point.

Given that we only have one *tap* point in each training sample, there will be only one out of all 256x512 points in the heatmap being set to 1, while all other points are set to 0. Therefore, our

dataset has a similar data imbalance problem as encountered in many object detection tasks [18]. We used two strategies to alleviate this issue. First, instead of setting only one point in the heatmap as 1 and others as 0, we found the bounding box of target UI element (Section 4.1.1) and then applied the 2D Gaussian function to obtain the probability of surrounding points in the interaction elements [8, 15, 20]. Second, we used a variant of the focal loss [15, 18] to perform a weighted penalization on the low confidence data: let p_{ij} be the predicted score (a.k.a. confidence) at location (i,j) in the predicted heatmap, and let y_{ij} be the ground-truth score augmented by the
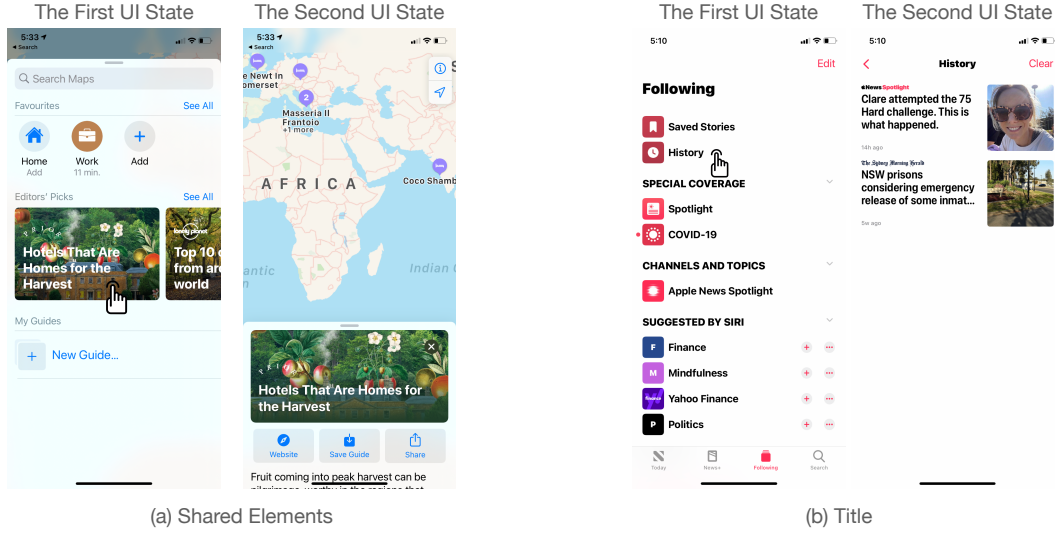
(a) Shared Elements   (b) Title

Figure 4: Examples of patterns that our interaction localization heuristics look for to localize a *tap* interaction, including (a) showing the start and the end UI state when the user taps on an item, and the item remains in the next page having the same label and the same image, and (b) showing an example when the tapped item's label becomes the title of the next UI state.
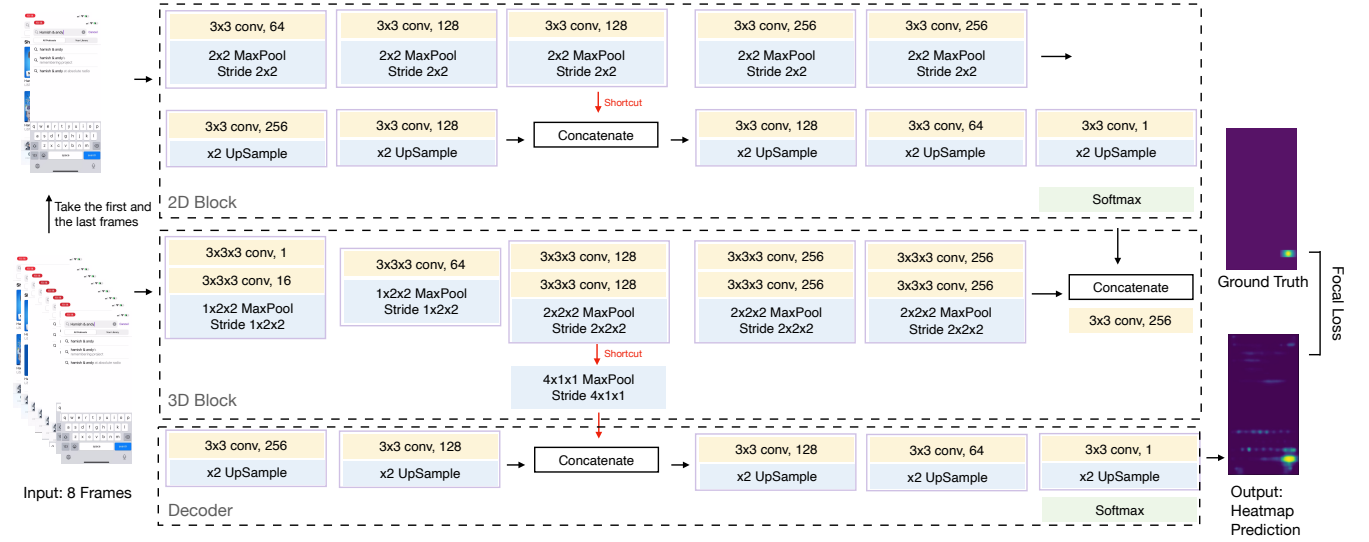

Figure 5: The structure of our interaction localization model. Given eight frames as the input, the model takes the first and the last frames as the input to the 2D block to learn the connections between the two consecutive UI states; Concurrently, the model feeds all eight frames into a 3D block to encode the animation effect; the model later concatenates the extracted features from the 2D block with the animation features extracted from the 3D encoder, and the combined features are then feed into a decoder to predict the interaction heatmap.

2D Gaussians. Then, the loss function is:

$$L = -\frac{1}{N} \sum_{i=0}^{H} \sum_{j=0}^{W} \begin{cases} (1 - p_{ij})^{\alpha} log(p_{ij}) & if\ y_{ij} = 1 \\ (1 - y_{ij})^{\beta}(p_{ij})^{\alpha} log(1 - p_{ij}) & otherwise \end{cases}$$

We used the interaction trace and animation datasets in Rico dataset [7] to train our model. The details are explained in Section 4.1.1. Our localization model is trained on 4 Tesla V100 GPUs

using an Adam optimizer [14] for 25 epochs with the initial training rate being 0.01 and a batch size of 32. For each interaction clip, we evenly picked 8 frames from the interaction clips as model input. We can also pick more frames as input, and we evaluated its impact in Section 4.4 Therefore, the structures of 3D blocks for these two inputs are slightly different, with the 3D pooling layer in the third convolutional block having a depth stride of 1 or 2. If the interaction

clip does not contain 8 frames, we duplicate some of the frames to get 8 frames. During training, the first and the last frame is fed into the 2D block and extract features from the first and the next UI state. In parallel, all frames are fed into the 3D block to extract the semantics of the animation effect. The two extracted features from 2D and 3D blocks are then concatenated and fed into a decoder to predict the interaction heatmap. The original size of Rico animation clips are 281x500 (width x height), and we resized them to 256x512.

3.4 Phase 4—Interaction Replay

The previous phases extracted interactions from video. When replaying interactions on another device, we sometimes can directly repeat interactions on screen (for example, typing entered text), but otherwise need to find a matching target UI elements to apply the interactions—such as a UI element to tap or a point to start swiping.

To accomplish interaction replays, we run an object detection model [30] to detect UI elements on each keyframe of recorded video, and then find the UI detection that contains a *tap* point or a *swipe* initiation point; if there are multiple detections, we pick the smallest detection. To find the target UI on the screen of a new device, we run fuzzy matching [23] for text elements and leverage template matching [26] for non-text elements.

Specifically, we choose the text with the highest weighted ratio (case-insensitive, ignore punctuation) [23] as the matching target text element. For non-text element, we use it as the template image, and slide it over the new screenshot to find a location with the highest matching value. We used normalized correlation coefficient as our matching function. When the replaying devices have different resolutions than the recorded video, we scale (50% to 200%) accordingly to the template image so that its size is similar to the target UI element in new screenshot—that is, multi-scale template matching.

Running matching algorithms on every pixel of screenshot can be time-consuming. To speed up matching, we first limit the search space to the detected UIs on the new screen, and run matching on full screenshot only when we fail to find a match from UI detections.

4 EVALUATION

We evaluated our system on a large-scale dataset (Rico [7], created 4 years ago), and a smaller app usage recording dataset (iOS and Android versions of 64 top-downloaded apps) we collected and annotated recently (Section 4.1). We evaluated each phase of system: video segmentation (Section 4.2, iOS–84.7%, Android–72.0% recall), interaction classification (Section 4.3, iOS–87.6%, Android–89.3% accuracy), interaction localization (Section 4.3, Rico–69.1%, Android–56.2%, iOS–41.4% accuracy), interaction replay across devices (Section 4.5, iOS–84.1%, Android–78.4% success rate).

4.1 Datasets

4.1.1 Interaction Clips from Rico Dataset. The Rico dataset [7] is a large-scale repository of Android app screens. In addition to UI element information on each screen (e.g., bounding box, UI class), the dataset also contains interaction traces of the apps and their corresponding video clips—for example, displaying animations after performing each interaction.

Each interaction trace provides a list of gestures to perform interaction, and we needed to derive an interaction type and interaction point from each gesture. We consider six interaction types as in [17], namely *tap*, *swipe left*, *swipe right*, *swipe up*, *swipe down* and *type*. We adopted the following heuristics from [17] to determine *tap* and *swipe* interactions:

(1) If an interaction contained only one gesture point or the distance of the gesture was ≤ 10 pixels, we considered it a *tap* interaction.

(2) If an interaction contained a list of gesture points with distance > 10 pixels, we considered it a *swipe* interaction. We mapped the gesture direction to *swipe left*, *swipe right*, *swipe up*, or *swipe down*.

For the *type* interaction, we noted that Rico dataset workers used physical keyboards to type text, and the *type* interactions were not recorded in the gesture data as a result. From our observations, we noted that the *type* interactions happened after a *tap* interaction on text field. Therefore, we detected these *tap* interactions and text changes in the text field, and then manually verified these potential *type* interactions.

In total, we obtained 44,536 interactions (with interaction type and video clip) from 7,211 user interaction traces in 6,547 free Android apps. Among these interactions, 91.7% (40,855/44,536) are *tap*, 0.3% (123/44,536) are *type*, 5.2% (2,299/44,536) are *swipe up*, 1.0% (442/44,536) are *swipe down*, 1.54% (688/44,536) are *swipe left*, and 0.3% (129/44,536) are *swipe right*. We found several limitations in Rico interaction traces and their video clips. Because the dataset only contains clips—and not a continuous usage recording—we were unable to evaluate our segmentation method using Rico. Some interactions omit their video clip or gesture, while other gestures do not match the video clips. Some video clips contain no changes in the UI. These data quality issues may significantly impact interaction classification result (especially on less frequent classes), as the interaction types are already highly skewed. Nevertheless, we were able to evaluate Rico on our interaction localization model—as our identified data issues have negligible impact for *tap* interactions. Thus, we used the Rico *tap* interactions to train our interaction localization model, and report our model performance on this testing split. Note that as we only use *tap* data to train the localization model, so that the localization model will not be biased.

An additional limitation of the Rico dataset is that it contains only Android apps, and was collected four years ago. As a result, this dataset may not reflect recent app designs on major mobile platforms. Thus, we collected and annotated usage recordings from top-downloaded iOS and Android apps—as discussed in the next section.

4.1.2 Usage Recordings from Top-Downloaded iOS and Android Apps. We followed the process of Bernal-Cárdenas et al. [2] to collect our dataset, and ensure its diversity and representativeness. We collected 64 top-downloaded free apps from the 32 categories in the Australia Google Play store (two apps per app category), all of which also offered a free iOS version to enable fair evaluation between different platforms. To ensure the collected recordings were representative, all authors discussed and selected the tasks, which include the key features of each app based on their description in

Table 2: The total number of interactions for each interaction type, and average task duration across 128 collected recordings from 64 top-downloaded applications.

	#Taps	#Types	#Swipe-Ups	#Swipe-Downs	#Swipe-Lefts	#Swipe-Rights	#Total	Avg. Duration
Android	396	33	78	15	6	6	534	35.8s
iOS	391	36	74	12	3	2	518	29.3s
Total	787	69	152	27	9	8	1,052	32.6s

the both app stores.[2] The first and second authors, one female and one male, both without any disabilities, randomly picked an app, installed it on both an iPhone 11 (iOS 14, physical device, 1792 x 828) and Nexus 6P (Android 11, emulator, 2560 x 1440), and recorded the screen while performing the same task in both the iOS and Android apps. When recording the videos, the two authors used the mobile apps as normal with no restrictions on their interactions.

Once app usage videos were recorded, the first author manually annotated them to segment stable intervals, classify interaction types, and locate interaction elements. We used an open-source tool, labelImg [27], to facilitate the annotation process. In total, we obtained 128 app usage recordings (μ duration = 32.6 seconds), containing 1,052 interactions (787 *taps*, 69 *types*, 196 *swipes*). Additional details are found in Table 2).

4.2 Phase 1—Video Segmentation

We evaluated our model's performance in video segmentation on usage recordings from top-downloaded iOS and Android apps. We examined each keyframe predicted by our model with all stable intervals in annotated ground truth. We classified our video segmentation are correct when a predicted keyframe falls into a stable interval (with no other predicted keyframes in this interval). We counted the # of correctly predicted keyframes (C), # of predicted keyframes (P), and # of annotated ground truth keyframes (A), and then calculated precision ($\frac{C}{P}$), recall ($\frac{C}{A}$) and F1-score.

Table 3 shows the performance of the video segmentation phase using different features and feature distance functions. Among all combinations, YUV+L1 performs the best (67.9% F1 score) on Android recordings while HOG+SSIM (81.9% F1 score) performs the best on iOS recordings.

Among all features, color histogram performed relatively the worst as it simply aggregates the general image features and somewhat downplays the salient changes. RGB and YUV features performed similarly as they essentially describe the same features with different representations. The HOG feature achieved the best recall (72.0% in Android recordings, and 84.7% in iOS recordings), which suggests that it effectively captures UI changes.

Among all feature distance functions, L2 distance had the worst performance, as it may overemphasize large changes. However, a distinguishable change does not necessarily imply a change in UI states. For example, changes in advertisement banner should not be considered as a new UI state. SSIM had the best performance, as it is a perceptual metric, which is able to capture general information about the image.

Table 3: Experimental results for video segmentation on recordings from top-downloaded apps on iOS and Android. We report Precision (P), Recall (R), and F1 score for each combination of feature extraction method and distance function.

	Android			iOS		
	P	**R**	**F1**	**P**	**R**	**F1**
RGB + L1	67.1%	68.3%	67.7%	**80.1%**	77.5%	78.7%
RGB + L2	49.7%	47.4%	48.5%	64.8%	74.0%	69.1%
RGB + SSIM	62.0%	70.4%	65.9%	78.5%	**84.7%**	81.5%
YUV + L1	**67.4%**	68.3%	**67.9%**	80.0%	77.8%	78.9%
YUV + L2	50.9%	51.3%	51.1%	68.3%	68.2%	68.2%
YUV + SSIM	62.2%	69.6%	65.7%	78.5%	83.5%	80.9%
Hist + L1	61.2%	66.2%	63.6%	79.8%	76.1%	77.9%
Hist + L2	62.5%	65.2%	63.8%	79.0%	72.3%	75.5%
Hist + SSIM	51.4%	58.7%	54.8%	70.0%	52.2%	59.8%
HOG + L1	57.6%	71.1%	63.6%	76.6%	83.1%	79.7%
HOG + L2	56.5%	56.4%	56.4%	73.8%	83.3%	78.2%
HOG + SSIM	61.0%	**72.0%**	66.0%	79.2%	**84.7%**	**81.9%**
SIFT	52.5%	58.4%	55.3%	70.3%	71.6%	71.0%

Overall, our method can effectively segment app usage videos into interaction clips when UI states have salient differences. We would like to share insights when our method fails to predict a keyframe. We missed keyframes when user interactions lead to a subtle change (or no change) on the UI. For example, when users select an item in a list, a small checkmark will appear. Such a small difference may be ignored by our simple feature extraction methods. Understanding the whole screen context would help us capture this important change on the UI. The effect of these errors are they reduce recall. We predicted extra keyframes on animations that are not caused by user interactions. For example, when users enter an image-heavy screen, a loading animation may appear while waiting. Similarly, when users download a file, a progress bar updates frequently and may automatically move to the next UI state once the file is downloaded. In the future, we should recognize these common animations. The effect of these errors reduce precision.

We also obtained insights from the performance differences we found between iOS and Android recordings. Our method may predict frames with changing advertisements as extra keyframes, which are not caused by user interactions. From our observations, Android apps tended to contain more banner advertisements while iOS apps displayed fewer advertisements. In addition, the Android emulator we used may have higher latency than their physical

Table 4: Experimental results for interaction classification on recordings from top-downloaded apps on Android and iOS, including Precision (P), Recall (R), and F1 Score

	Android			iOS		
	P	**R**	**F1**	**P**	**R**	**F1**
Tap	94.6%	91.9%	92.7%	94.6%	89.2%	91.8%
Type	74.4%	87.9%	80.6%	57.1%	100.0%	72.7%
Swipe Up	92.9%	81.2%	86.7%	90.2%	74.3%	81.5%
Swipe Down	46.7%	53.8%	50.0%	64.3%	75.0%	69.2%
Swipe Left	46.2%	100.0%	63.2%	33.3%	100.0%	50.0%
Swipe Right	75.0%	100.0%	85.7%	100.0%	100.0%	100.0%
Macro Avg.	71.4%	85.8%	76.5%	73.2%	89.8%	77.5%
Weighted Avg.	90.4%	89.3%	89.6%	90.3%	87.6%	88.3%
	Accuracy: 89.3%			Accuracy: 87.6%		

Table 5: Accuracy of our interaction localization model compared with several baselines for the Rico-Test dataset and our manually collected recordings.

	#Frames	Rico-Test	Recordings	
			Android	**iOS**
Humanoid	8/16	29.5%/29.5%	8.8%/8.8%	9.7%/8.7%
HM2D	8/16	61.8%/52.4%	34.3%/21.7%	28.1%/21.1%
HM3D w/o shortcut	8/16	66.7%/66.9%	52.8%/53.0%	36.2%/38.9%
HM3D	8/16	67.9%/67.0%	53.0%/54.3%	40.4%/38.5%
HM3D+2D	8/16	**69.1%/67.9%**	52.3%/54.3%	41.0%/40.0%
HM3D+2D+Heuristics	8/16	**69.1%/67.9%**	**53.6%/56.2%**	**41.4%/40.4%**

counterpart devices. Because the emulator takes longer for UI rendering and UI transitions, this makes it harder to distinguish UI rendering and transitions from user interactions.

4.3 Phase 2—Interaction Classification

We evaluated our model's performance in interaction classification on usage recordings from top-downloaded iOS and Android apps.

Table 4 shows the performance of the interaction classification phase, which performs well on both iOS and Android app recordings (87.6% and 89.3% accuracy respectively). Our method achieves high recall in most interaction types (except *swipe down* on Android), and gets high F1 scores in *tap*, *swipe up*, and *swipe right.*

Swipe left and *swipe down* had the worst performance. As they have only 9 and 27 samples out of 1,052 interactions, their precisions can be impacted by incorrect predictions from the other interaction types. Here is an example of failures of *swipe left*: the screen scrolls horizontally when users *tap* on the next segmented control, which has the same visual effect of *swipe left*. We also found that half of *swipe down* interactions on Android were recognized incorrectly as *tap*. Most of these failures were related to a date/time picker: text in the pickers are smaller and faded to highlight currently selected text, which reduced accuracy of OCR that our heuristics rely on. Some failures in *swipe down* happen when users *tap* a button in the bottom actionsheet; the actionsheet moves down and disappears, creating a similar visual effect of *swipe down*. Not surprisingly, we also found that most false positives are from *tap*—the majority of interactions in our dataset.

Our method had reasonable performance on *type* interactions. However, when the virtual keyboard appears, users may still perform non-type interactions like *tap*. In the future, we should focus on the visual changes inside the virtual keyboard to confirm *type* interactions.

4.4 Phase 3—Interaction Localization

We evaluated our model's performance in interaction localization on the large-scale Rico dataset, and usage recordings from top-downloaded iOS and Android apps.

Table 5 shows the interaction localization performance of our system, and several baseline methods as comparison. The first baseline is **Humanoid** [17], which predicts the next interaction given the previous three UI screens. It follows a RNN-style method to encode frame features step-by-step, and predicts the heatmap of possible interaction points using a decoder module. There are also variants of our model as baselines: **HM2D** (shorts for heatmap 2D) directly uses 2D convolutional layers to learn the semantics from animations, while **HM3D** instead uses 3D convolutional layers to learn the temporal and spatial features from animations through several layers. The default **HM3D** contains a UNet-style shortcut, which is expected to help the model refine the high-level abstract features using the features from shallow layers. We also consider a variant of **HM3D without shortcut** to see the impact from the shortcut module. Another variant close to our model is **HM3D + 2D**, while our final system (**HM3D + 2D + Heuristics**) includes heuristics to improve performance. We also compared the performance when the input contained 8 or 16 frames from interaction clips.

4.4.1 Performance on Rico Dataset. Our system outperformed all baselines, reaching 69.1% accuracy when the input contains 8 frames from interaction clip. We found the 3D convolutional network (HM3D) better captured the temporal features from interactions than RNN-style (Humanoid) and 2D convolutional based network (HM2D). RNN-style model encodes each frame, and the compressed frame may lose information before it is fed into the next RNN cell. A 2D-style model heavily relies on the first layer to capture the temporal features among frames, while a 3D-style model gradually learns the semantics of animations through several layers. UNet-style shortcut and additional 2D modules both help our model to better learn features and slightly boosted the performance. Applying heuristics also improved model performance in recent app recordings.

We then analyzed the performance across UI element types as they have different visual effects. We considered six common UI types in Rico dataset, namely: ImageButton, ImageView, TextView, Button, and System Bottom Navigation Bar. The rest of UI elements fall into "Other" type. From Table 6, we found TextView and ImageView elements had worse performance compared to other UI types. Other UI types are tappable by default and they have animations provided by the system UI framework. TextView and ImageView are not tappable unless developers specify the property or create a customized event listener. Therefore, these two UI types are more likely to have a special animation effect or no

Table 6: The recall of our interaction localization model as compared with several baselines, across 6 common UI types in the Rico Dataset. The table shows the results for models trained on 8 frames / 16 frames.

	Text	Button	ImageView	ImageButton	System Nav. Bar	Others
Number	*783*	*625*	*521*	*531*	*388*	*1,592*
Humanoid	1.1%/1.3%	3.5%/3.7%	17.1%/17.1%	68.7%/68.7%	N/A	51.6%/51.6%
HM2D	33.3%/20.1%	63.8%/51.7%	44.7%/34.5%	79.3%/77.4%	82.0%/62.6%	69.5%/62.8%
HM3D w/o shortcut	43.0%/43.3%	68.6%/66.7%	45.9%/46.8%	83.4%/84.4%	87.4%/90.2%	72.7%/72.5%
HM3D	44.3%/40.2%	70.4%/67.4%	48.9%/49.5%	83.1%/85.3%	89.2%/89.9%	74.1%/73.3%
HM3D + 2D	44.8%/43.3%	70.9%/69.9%	49.1%/47.4%	85.5%/86.1%	89.4%/89.9%	75.4%/73.6%

visual effect. In contrast, all models performed best on the system back button (except for Humanoid) and ImageButton, which almost always provide visual feedback when users *tap* them.

4.4.2 Performance on Recent iOS and Android App Recordings. Our model is trained on the Rico dataset collected 4 years ago, and we wanted to investigate the feasibility of our method on recent mobile apps. As shown in Table 5, we found that the performance of all models degrade.

Since the release of the Rico dataset, design principles and UI styles have changed substantially in recent mobile apps. One example is the redesign of system bottom navigation bar. Previously, Android apps avoided using the tab bar at the bottom of the screen (side menu drawer is a replacement), because users could easily tap system bottom navigation bar by mistake. Nowadays, the system bottom navigation bar no longer shows buttons, but only a subtle bar with space to enable gesture navigation. Android apps are more likely to use the tab bar instead of the menu drawer. From the Rico dataset, we randomly selected 100 apps and sampled one interaction from each app. Only 6% of apps contain a bottom tab bar, while most of the recent iOS (87.5%) and Android (75%) apps we collected contain a bottom tab bar.

We also found the animation visual feedback to be more subdued now. For example, the text color may only slightly change after a *tap*. After examining the failure cases in recent app recordings, we found that our model worked best when the animation visual feedback is more apparent. As seen in the first two keyframes of Figure 2, users *tap* on the bottom tab bar—which leads to a subtle change in the text color of a tab button and an obvious change in the main content. In the future, the understanding of all UIs on a screen [30] will help our model focus on important UI changes—for example, to prioritize tab button changes when a tab bar is detected.

Finally, we noticed a large performance discrepancy between iOS and Android recordings, as the differences in their designs are even more substantial than the differences between Rico and recent Android apps. A larger-scale app usage recording dataset in both iOS and Android, like Rico, will help our model better capture the interactions under these new UI paradigms.

4.5 Phase 4—Interaction Replay

We evaluated our model's performance in interaction replay on usage recordings from top-downloaded iOS and Android apps. In order to evaluate the success rate of interaction replay, the first two authors also collected the same app interaction traces on devices with different resolutions (Pixel 4 XL running Android 11 and

iPhone 11 Pro Max running iOS 15). It is a manual replay process that will not stop by error in one step.

To focus on the performance of replay module itself, we directly used the annotated interactions as a ground truth to avoid the errors that propagate from each step during interaction extraction.

During this new collection, we took notes of four problems that prevented us from replaying 23 interactions on the target devices. First, some pop-ups windows appear occasionally while replaying and required us to perform extra steps to close them. These pop-up windows include advertisements, instruction hints, rating requests, and permission requests. Second, some interactions were related to specific time that were no longer available. For example, when we try to replay the interaction in a different month, the option of previous month may no longer exist and thus we cannot replay the exact same interaction. Third, apps may contain dynamic content that changes the required user interactions. For example, in a recording, we need to *swipe up* five times to reach the target element to *tap*; in the updated content, we only need to *swipe up* twice. Fourth, apps add and remove features in updates. Such updates could lead to changes in the UI layout and UI transitions, removing an existing UI element, or affecting the navigation logic. We counted the cases of each problem during the replay process. Among failure cases in iOS | Android recordings, 9 | 12 are relevant to the pop-up windows, 1 | 1 are relevant to the time, 6 | 5 are relevant to dynamic content, and 3 | 5 are relevant to app updates.

For each interaction, our system found a matching target UI element on newly collected screens. The first two authors manually examined the matching result to determine whether interactions on the matched UI could lead to the expected next UI state. The majority of interactions could be correctly replayed in iOS and Android apps (84.1% and 78.4% respectively).

Here are some common failure cases in UI element matching: Image content may change across different sessions or change due to personalization (e.g., albums in Figure 6(a) are different in different accounts). Therefore, our image template matching would not find the same image. There can also be multiple UI elements contain the same text, and our text matching may find the wrong target text element. Beyond the scope of this paper, a deeper understanding of UI will help in resolving these failure cases: after our system learns what content is dynamic and what content is repetitive, it can replay the interaction on the UI element that has the same relative position in the UI structure.

5 DISCUSSION

Datasets. We evaluated our proposed system with two different datasets. First, we use the Rico dataset to evaluate our localization models in a large-scale experiment, even though the Rico dataset only contains interactions with Android apps. The addition of a large-scale interaction dataset of iOS apps would help better illustrate the advantages and disadvantages of our system. To mitigate this issue, we then collected interaction traces from two top-downloaded apps from each app category that were available on both the iOS and Android platforms. We used these traces to better evaluate the generalizability and robustness of the proposed system. To ensure the manual recordings were representative, all authors together discussed and selected the tasks to be performed, ensuring that they covered the key features of all apps based on their descriptions in the app stores. While each type of interaction has a different number of trials, we believe the diversity and representativenss of the collected apps and interactions mitigates some of the potential issues. We also report the detailed results for each interaction to better illustrate the performance on different interactions. Our future work will include more data to better examine and improve our system. Moreover, in the future we will examine extending our work to other input devices of different resolutions, such as larger-screened tablets.

Opportunities for performance improvements. Our approach has several opportunities for performance improvements. First, several limitations in detecting interactions were a result of modern UI interaction paradigms which were not available when the Rico dataset was released. Consequently, we expect that a large-scale dataset of *recent* apps on multiple mobile platforms would improve our system performance. Such a dataset would provide relevant data to train machine learning models to enhance our heuristic-based video segmentation and interaction classification phases, as well as and make our interaction localization model more generalizable to apps on iOS and Android platforms.

Our current localization model relied on video frames and their corresponding pixel-data, achieving around 70% accuracy. This localization model could be improved in several ways. First, we could consider running UI detection on a screen to predict the interactable UI elements, as in Chen et al. [5], Zhang et al. [30]. These UIs have higher probabilities for *tap* interactions. Additional labels [3, 4] for some image-based interactable elements can also improve the system. Second, we tried a simple heuristic-based method to identify the connections between *tapped* text and the title in new UI state. Deeper understanding of the content of text elements would support inferring the interaction between the two UI states. For example, ActionBert [13] demonstrates that it is possible to predict connection elements between two UI states—even without leveraging animation.

We proposed a straightforward method to replay interactions. In our evaluation, it works well on the same (or similar) app versions for a different device. This assumption applies to some applications— for example, automated or regression testing for the the same app version—but other scenarios, such as making app tutorials, may require using multiple app versions. Another challenge is to replay the interactions for the same app in different languages, which would enable cross-locale applications. Collecting the interaction

traces in different settings (e.g., app versions, languages) would provide more signals for our interaction replay phase. Our current system only tests recordings within one app, while many tasks involve multiple apps. Learning the transition between apps will enable cross-app interaction extraction and replay to complete more complicated tasks.

We think of our pixel-based approach as a general technique that is also in some ways a *lower-bound* on accuracy: by design, it does not take advantage of additional metadata that could potentially further improve its performance. Incorporating metadata, if available—such as the UI framework used within the app, platform, and version—could boost the performance of our approach. Of course, the disadvantage of metadata is that it is not always available, difficult to extract, or unnecessarily constraints and couples the model to the metadata.

Applications of extracting replayable interactions. There are several applications that may benefit from our methods to extract and replay user interactions from video pixels. A straightforward application is to allow users to *annotate interactions on existing videos*. For example, they may have a screen recording and have difficulty figuring out how the user in the screen recording is getting to a particular screen. Our system could be used to provide on-screen annotations of our inferred interactions as the user plays the video.

For *app bug reports*, users or QA testers could create videos of issues in apps, which developers could then replay within their own development environment to reproduce. Similarly, end-users could upload videos to demonstrate app usage problems: automatically identifying the interactions in these videos could minimize or eliminate the errors introduced by more manual identification procedures. In *automated app testing*, QA testers can sometimes only run apps on unmodified devices, which do not allow special recording tools or collection of metadata. Our method extracts interaction traces from app usage videos, and then replays them on other devices to test. After collecting a larger-scale app dataset on multiple platforms, our pixel-based method could potentially enable cross-platform testing without relying on the platform-specific testing APIs.

Finally, our approach could be used in *app tutorials*. As one example, people with limited mobile usage experience and people with cognitive impairments sometimes require help from others (such as their caregivers) to use a new app or an updated version of app. Our method might be applied to automatically create app tutorials from app usage video recorded by users who better understand and can demonstrate the app functionality. Then, people in need can replay interactions on their own mobile devices, or learn how to use apps with an on-device, interactive tutorials [12].

6 CONCLUSION

In this paper, we introduced a novel approach to automatically extract and replay interactions from video pixels without requiring additional settings, recording tools, or source code access. Our approach automatically segments interactions from a video, classifies interaction types, and locates target UI elements for replay. We trained our system using the large-scale Rico dataset for Android, evaluated its effectiveness, and demonstrated the feasibility

of learning interaction locations for recent iOS and Android apps. Our prototype can successfully replay the majority of the interactions. The results of this work suggest that extracting replayable interactions is a useful mechanism that potentially benefits a variety of different applications and scenarios.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Towards Complete Icon Labeling in Mobile Applications. In *CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 387, 14 pages. https://doi.org/10.1145/3491102.3502073

[5] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: old fashioned or deep learning or a combination?. In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1202–1214.

[6] Navneet Dalal and Bill Triggs. 2005. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, Vol. 1. Ieee, 886–893.

[7] 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.

[8] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. 2019. Centernet: Keypoint triplets for object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 6569–6578.

[9] Sidong Feng and Chunyang Chen. 2022. GIFdroid: Automated Replay of Visual Bug Reports for Android Apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. ACM.

[10] Anhong Guo, Junhan Kong, Michael Rivera, Frank F Xu, and Jeffrey P Bigham. 2019. Statelens: A reverse engineering solution for making existing dynamic touchscreens accessible. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 371–385.

[11] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 215–224.

[12] Kyle J Harms, Jordana H Kerr, and Caitlin L Kelleher. 2011. Improving learning transfer from stencils-based tutorials. In *Proceedings of the 10th International Conference on Interaction Design and Children*. 157–160.

[13] Zecheng He, Srinivas Sunkara, Xiaoxue Zang, Ying Xu, Lijuan Liu, Nevan Wichers, Gabriel Schubiner, Ruby Lee, Jindong Chen, and Blaise Aguera y Arcas. 2020. ActionBert: Leveraging User Actions for Semantic Understanding of User Interfaces. *arXiv preprint arXiv:2012.12350* (2020).

[14] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[15] Hei Law and Jia Deng. 2018. Cornernet: Detecting objects as paired keypoints. In *Proceedings of the European conference on computer vision (ECCV)*. 734–750.

[16] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.

[17] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[18] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*. 2980–2988.

[19] Cuong Nguyen and Feng Liu. 2015. Making software tutorial video responsive. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 1565–1568.

[20] Tomas Pfister, James Charles, and Andrew Zisserman. 2015. Flowing convnets for human pose estimation in videos. In *Proceedings of the IEEE international conference on computer vision*. 1913–1921.

[21] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. Roscript: a visual script driven truly non-intrusive robotic testing system for touch screen applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 297–308.

[22] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[23] seatgeek. 2021. GitHub - seatgeek/fuzzywuzzy. https://github.com/seatgeek/fuzzywuzzy. Accessed: 24/09/2021.

[24] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. 2005. Automated replay and failure detection for web applications. In *Proceedings of the 20th IEEE/ACM international conference on automated software engineering*. 253–262.

[25] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. 2015. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*. 4489–4497.

[26] OpenCV-Python Tutorials. 2021. Template Matching. https://opencv24-python-tutorials.readthedocs.io/en/stable/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html. Accessed: 24/09/2021.

[27] tzutalin. 2021. GitHub - tzutalin/labelImg. https://github.com/tzutalin/labelImg. Accessed: 24/09/2021.

[28] 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.

[29] Shengcheng Yu, Chunrong Fang, Yang Feng, Wenyuan Zhao, and Zhenyu Chen. 2019. Lirat: Layout and image recognition driving automated mobile testing of cross-platform. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1066–1069.

[30] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[31] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O Wobbrock. 2017. Interaction proxies for runtime repair and enhancement of mobile application accessibility. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6024–6037.

[32] 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.

[33] Di Zhong, HongJiang Zhang, and Shih-Fu Chang. 1996. Clustering methods for video browsing and annotation. In *Storage and Retrieval for Still Image and Video Databases IV*, Vol. 2670. International Society for Optics and Photonics, 239–246.

7 APPENDIX

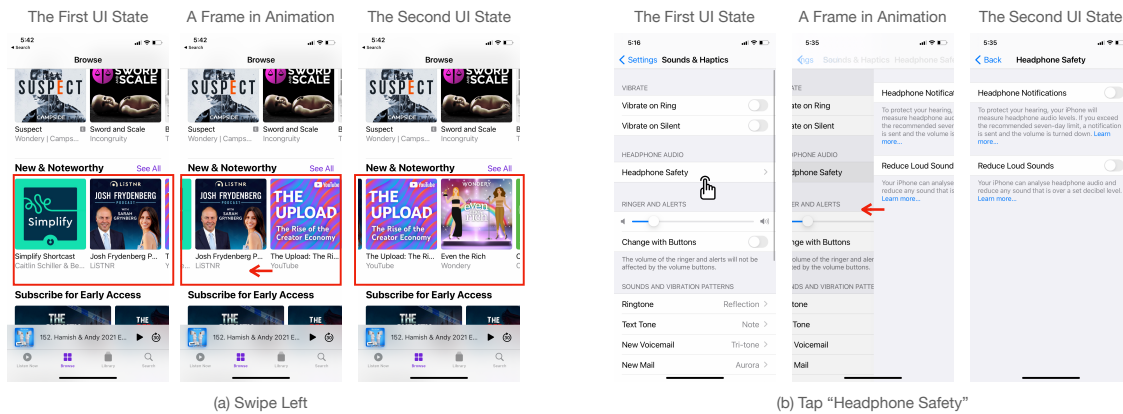(a) Swipe Left

(b) Tap "Headphone Safety"

Figure 6: Examples of patterns that our interaction classification heuristics examine to classify a *swipe left* interaction, including (a) a *swipe left* interaction will likely change at least 3 text elements, and will not change the UI title, and (b) a *tap* interaction instead is likely to change the title.
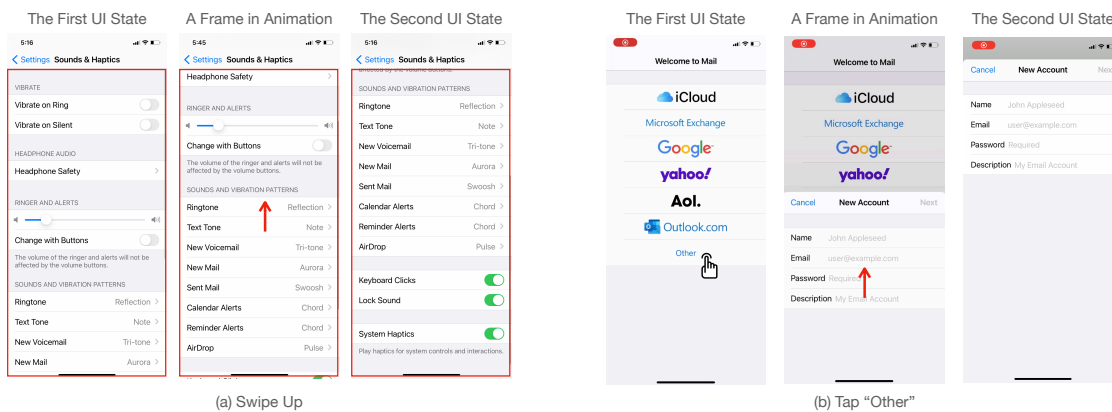

(a) Swipe Up

(b) Tap "Other"

Figure 7: Examples of patterns that our interaction classification heuristics examine to classify a *swipe up* interaction, including (a) a *swipe up* interaction will likely change several text elements, but will not change the UI title, and (b) *tap* interaction instead is likely to change the title.