RIDA: Cross-App Record and Replay for Android

Jiayuan Liang*, Sinan Wang†, Xiangbo Deng*, Yepang Liu*†
*Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
†Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China
{11910504,wangsn,11913006}@mail.sustech.edu.cn; liuyp1@sustech.edu.cn

***Abstract*—The number of Android apps keeps increasing in recent years. Despite the fact that there exist apps for various kinds of purposes, apps that share similar functionalities with existing ones are still emerging on the market. To reduce the effort in testing such apps, previous research has proposed approaches for migrating test scripts across similar apps. However, such test reuse techniques require existing test suites for migrating, which hinders their practical use in commercial app development. Unlike script-based GUI testing, record-and-replay techniques are more convenient for human testers who are unfamiliar with programming. In this paper, we propose a new testing technique, RIDA, that records interaction sequences on one app (source app) and replays them on another app (target app) with similar functionalities. Such *cross-app record-and-replay* is challenging. First, there are no clear mappings between the recorded sequences on source apps and the sequences required to be performed on the target apps. Second, reliable indicators of widgets' functionalities are not always available from the recorded sequences, which limits the effectiveness of event matching between source and target apps. To address the challenges, we design an on-the-fly searching algorithm for finding target widgets during cross-app replay and leverage multiple semantic descriptors together with image captioning techniques to infer the functionalities of widgets. We have implemented RIDA and evaluated it using both controlled and in-the-wild experiments. The results show that RIDA can effectively perform cross-app record-and-replay and outperform baseline methods in terms of the number of completely- and partially- replayed events.**

***Index Terms*—Android testing, cross-app record-and-replay, semantic matching, image captioning**

I. INTRODUCTION

According to Statista [1], Google Play, the official Android app market, possesses more than 2.6 million apps. They offer a plethora of functionalities to Android users. Despite the fact that Android app markets are growing and the app categories are enriching through the years, apps with similar functionalities to existing popular ones are still emerging. For example, at the time of paper writing, we searched the keyword "calculator" with a "New" filter on AppBrain [2], among the 487 resulting apps, 360 were published in the recent month according to their "App age" attributes. These apps can be diverse in their appearances [3], language preferences [4], or privacy policies [5], but their core functionalities are typically the same. For instance, mobile browsers must be able to visit a website by its URL or navigate between different web pages, regardless of their color schemes or widget shapes. Moreover, these core functionalities are usually triggered via particular user interaction patterns (e.g., enter a URL at the top bar of a browser to visit a new web page). Reusing common interaction patterns for specific functionalities is a good practice for reducing users' learning costs on newly installed apps [6].

On Android, developers usually write GUI test scripts using test automation frameworks, like UIAutomator2 [7], for quality assurance [8]–[11]. However, for a newly created app, developing and maintaining a test suite can be labor-intensive and expensive [12]. To reduce the cost of testing such apps, recent studies have proposed techniques for reusing existing test suites from other apps with similar functionalities. This approach is called *test reuse* [13]–[16]. However, existing test reuse techniques suffer from a major limitation: they require a test suite from the reference app as input, which is usually unavailable for large commercial and closed-source apps. This prevents the practical use of migrating Android GUI tests through test reuse.

Despite the abundant research on GUI test automation, manually performing GUI testing is still widely adopted by Android developers [17]. Given that automated test scripts are hard to maintain, *record-and-replay* becomes a good alternative to facilitate GUI testing [18]–[22]. In record-and-replay, human testers do not need to write test code. Instead, they will manually perform user events on the test device, and these events are recorded as test cases which will be replayed for regression testing purposes. Our insight is, in addition to regression testing, these recorded test cases can also be reused (or replayed) to test other apps with similar functionalities. We call such a scenario as *cross-app record-and-replay*, in which the test cases are recorded on a *source app*, and will be transferred to exercise another *target app*.

Existing record-and-replay approaches on the Android platform mostly concern the stability and compatibility of GUI test cases across different Android devices. These tools cannot be directly adapted to a cross-app scenario. There are two technical challenges for cross-app record-and-replay. First, as the source app and the target app are created by different developers, their ways of interaction are typically not identical. For the same usage scenario, their required user events can be different. Hence, cross-app record-and-replay should be able to map one event in the source app to multiple events in the target app, and vice versa. Second, unlike test reuse which identifies source event widgets through unique textual locators (i.e., the `resource-id` attributes) from the test scripts, the raw recorded events are typically coordinate-based. Since texts are essential for semantic mapping between widgets [23], lacking accurate textual indicators of functionalities [24] can further

Yepang Liu is the corresponding author.

increase the difficulty of identifying appropriate target widgets.

We propose RIDA, a tool for **R**ecord-and-replay **I**n **D**ifferent Android **A**pps. RIDA addresses the aforementioned challenges in the following ways. To achieve one-to-many and many-to-one event mapping, RIDA first identifies the source widgets through self-replaying and extracts the widgets' semantic descriptors from the source app's UI hierarchy. For widgets without explicit textual indicators of functionalities, RIDA generates their semantic descriptors using the *image captioning* technique [24]. RIDA employs an on-the-fly bidirectional search algorithm to analyze the source event sequence, and identifies appropriate target events via semantic matching. Considering that a widget's text information is typically short in length, RIDA leverages part-of-speech tagging to reduce noises caused by common words from the short texts to facilitate semantic matching. Experiment results on 25 popular real-world Android apps show that RIDA is effective in replaying both controlled and unspecified user events. Moreover, RIDA's semantic matching algorithm outperforms the state-of-the-art approach and other baseline methods.

In summary, we make four contributions in this paper:

1) To the best of our knowledge, we made the first attempt to address the technical challenges for cross-app record-and-replay, which can help Android app developers boost their development process when QA resources are limited.
2) We designed and implemented RIDA, a practical tool that employs an on-the-fly search algorithm and image captioning technique to realize record-and-replay across apps with similar functionalities.
3) We evaluated RIDA by both controlled experiment and in-the-wild experiment, showing that RIDA is effective in replaying both controlled and unspecified user event sequences. The results demonstrate RIDA's usefulness for cross-app record-and-replay tasks.
4) We made RIDA's source code and all experiment data publicly available for future research [25].

II. PRELIMINARIES

A. Semantic Matching & Word Embedding

An important step for migrating recorded events across apps is comparing the semantic similarity between widgets' textual descriptions. This is also called *semantic matching* for GUI events [23]. Many test reuse tools utilize semantic matching technique to match similar widgets in source apps and target apps [13], [14]. Two widgets' semantic similarity is mostly calculated with their associated textual information. Such information, called *semantic descriptors*, is obtained through various means, for example, the textual attributes defined in the GUI's front-end code. The extracted texts should typically be vectorized into numerical forms through *word embedding* techniques [26]–[28], from which the similarity scores can be computed. According to Mariani et al.'s work [23], Word Mover's Distance (WMD) [27] is the state-of-the-art word embedding model for semantic mapping in app test reuse. It measures the dissimilarity between two sentences by finding the minimum cumulative distance between embedded words from one sentence and that from the other.

TABLE I
IMAGE CAPTIONS GENERATED BY LABELDROID

		
"next"	"more options"	"clear query"

B. Image Captioning

Image captioning is a technique that generates textual descriptions for visual images. Generally, image captioning models consist of two deep neural networks: a convolutional neural network (CNN) that extracts image features as one-dimensional numerical vectors, and a recurrent neural network (RNN) that generates descriptive sentences given the vectors returns from the CNN. Different image captioning techniques vary in the CNN and RNN structures they applied [29].

In mobile app development, image captioning can be used to improve app's usability and accessibility for users with visual impairment [30]. This task is realized by predicting textual labels (i.e., descriptors) for image-based widgets on mobile GUIs and reading them out via the devices' screen readers. Table I shows three examples of app buttons and their corresponding image labels generated by LABELDROID [24], a label prediction model for image-based buttons.

III. APPROACH

A. Overview

Figure 1 shows RIDA's workflow. Similar to many record-and-replay techniques [21], [31], it consists of three phases: (1) *Recording*, (2) *Self-Replaying*, and (3) *Cross-App Replaying*. Given a source app and a target app, RIDA records the actions performed by the human tester on the source app and replays the recorded test cases on the target app by searching for appropriate events and exercising them.

In the recording phase, the tester interacts with the source app. RIDA will record these actions and save the recorded input events into a log file in a raw coordinate-based form. In the self-replaying phase, RIDA replays the categorized input events on the source app to obtain the involved widgets and their semantic descriptors. In order to obtain reliable descriptions of the widgets' functionalities, RIDA leverages the UI hierarchies and screenshots of the source app to generate short captions from the widget icons. The categorized events, as well as their semantic descriptors, will be saved and used in the next phase. Finally, in the cross-app replaying phase, based on the recorded events, RIDA will identify the appropriate actions for the target app by performing semantic matching of events using the descriptors obtained from both apps and replay the actions on the target app. The following parts will describe the three phases in detail.

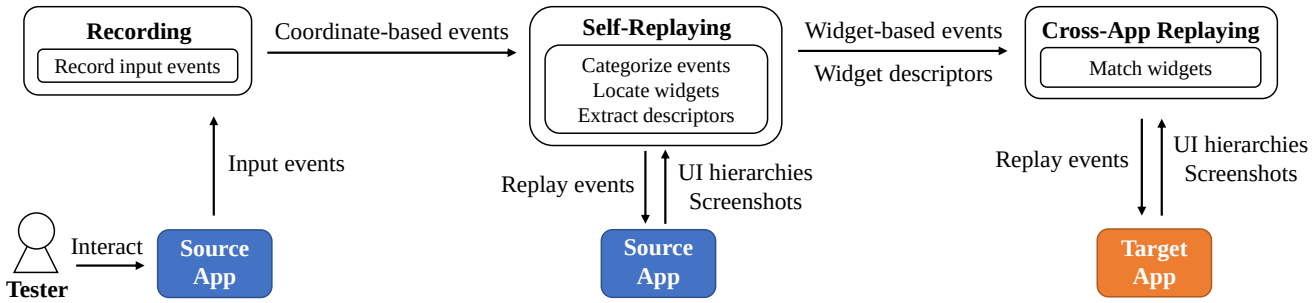


Fig. 1. The workflow of RIDA

B. Recording

The recording phase intercepts and dumps the tester's raw actions into a text file. RIDA utilizes the `getevent` command provided by the Android Debug Bridge (ADB) to capture user inputs [32]. Figure 2 gives a sample output of the `getevent` command with the options `-tl`, when two click events were performed on an Android device's touchscreen. The option `-l` means to display readable textual labels of the event codes and `-t` means to display timestamps. Each line can be divided into three fields. The first field is the timestamp in seconds since the system starts. The second field is the type of the input devices, such as phone cameras, accelerometers, etc. In the example, `/dev/input/event2` corresponds to interactions from the touchscreen. The third field provides detailed information about the input event as a 3-tuple. For instance, the 3-tuple ⟨`EV_KEY`, `BTN_TOUCH`, `DOWN`⟩ at line 1 is an event of touchscreen press, while the 3-tuple ⟨`EV_KEY`, `BTN_TOUCH`, `UP`⟩ at line 9 indicates the finger release. There can be more 3-tuples between these two events, which contain precise user interaction information. For example, line 3 records a `ABS_MT_POSITION_X` element with a hexadecimal value of `3e5`, which means the touch event at line 1 had an X coordinate of 997 on the device's screen. Similarly, line 4 indicates that the Y coordinate is 1795. During the recording phase, the tester performs actions on the Android device without intervention. RIDA saves the coordinate-based events until the tester finishes the recording phase.

	Timestamp	Device Type	Event Information		
1.	[406538.849065]	/dev/input/event2:	EV_KEY	BTN_TOUCH	DOWN
2.	[406538.849065]	/dev/input/event2:	EV_ABS	ABS_MT_TRACKING_ID	000013c3
3.	[406538.849065]	/dev/input/event2:	EV_ABS	ABS_MT_POSITION_X	000003e5
4.	[406538.849065]	/dev/input/event2:	EV_ABS	ABS_MT_POSITION_Y	00000703
5.	[406538.849065]	/dev/input/event2:	EV_ABS	ABS_MT_PRESSURE	0000000d
6.	[406538.849065]	/dev/input/event2:	EV_SYN	SYN_REPORT	00000000
7.	[406538.887656]	/dev/input/event2:	EV_ABS	ABS_MT_PRESSURE	00000000
8.	[406538.887656]	/dev/input/event2:	EV_ABS	ABS_MT_TRACKING_ID	ffffffff
9.	[406538.887656]	/dev/input/event2:	EV_KEY	BTN_TOUCH	UP
10.	[406538.887656]	/dev/input/event2:	EV_SYN	SYN_REPORT	00000000
11.	[406539.810633]	/dev/input/event2:	EV_KEY	BTN_TOUCH	DOWN
12.	[406539.810633]	/dev/input/event2:	EV_ABS	ABS_MT_TRACKING_ID	000013c4
13.	[406539.810633]	/dev/input/event2:	EV_ABS	ABS_MT_POSITION_X	0000029a
14.	[406539.810633]	/dev/input/event2:	EV_ABS	ABS_MT_POSITION_Y	0000065d
15.	[406539.810633]	/dev/input/event2:	EV_ABS	ABS_MT_PRESSURE	00000007
16.	[406539.810633]	/dev/input/event2:	EV_SYN	SYN_REPORT	00000000
17.	[406539.856073]	/dev/input/event2:	EV_ABS	ABS_MT_PRESSURE	00000000
18.	[406539.856073]	/dev/input/event2:	EV_ABS	ABS_MT_TRACKING_ID	ffffffff
19.	[406539.856073]	/dev/input/event2:	EV_KEY	BTN_TOUCH	UP
20.	[406539.856073]	/dev/input/event2:	EV_SYN	SYN_REPORT	00000000

Fig. 2. Sample raw input events of two clicks

C. Self-Replaying

The goal of the self-replaying phase is to locate the source widgets on which the tester's actions were performed and extract the widgets' semantic descriptors. Taking the file from the previous phase, RIDA will replay the recorded events in sequential order on the source app running on the same device. Before exercising an event, RIDA obtains two essential pieces of information from the source app. First, it obtains the *UI hierarchy*, i.e., an XML tree that represents the current app state. App widgets and their textual attributes are stored as nodes of the XML tree so that we can locate the interacted widgets through tree traversal and extract semantic descriptors from the attributes. Second, when the textual attributes are missing [24], RIDA will take a screenshot and crop the widget's image according to its bound denoted by the XML attributes. The cropped images will be used to generate additional descriptors. It is worth mentioning that the extracted semantic descriptors are device-independent. Thus the cross-app replaying phase can also be conducted on a different Android device.

1) Event Categorization: Similar to previous work on test reuse [13], [14], [23], RIDA supports three common types of touchscreen events: click, long press, and swipe. We refer to the official guidelines of Android `GestureDetector` [33] and `ViewConfiguraion` [34] utilities to categorize user events. Specifically, a continuous touch that wanders over eight pixels on the screen is considered a swipe. If the moving distance is within eight pixels and the duration is more than 400 milliseconds, it is regarded as a long press; otherwise, we consider it a click. Note that these rules also support text input events, in which a series of clicks are performed on the system's soft keyboard.

2) Widget Locating: RIDA locates the interacted widgets by referring to the `bound` attributes in the UI hierarchy. If an interacted coordinate is within a widget's rectangular bound, RIDA regards it as the interacted widget. Occasionally, there can be multiple widgets at the same coordinate. In such cases, we adopt a heuristic strategy to identify the interacted widget. Specifically, RIDA drops out widgets covered by the newly appeared widget after performing the event and chooses the one that occupies the least screen size. Our intuition is that

smaller widgets are more likely to be on top of other widgets on the screen, thus intercepting user interactions.

After locating the interacted widget, RIDA will determine its widget type. RIDA considers four widget types: editable app widget, non-editable app widget, soft keyboard widget, and system widget. Editable app widgets are instances of the class `EditText` [35] and its subclasses. Other app widgets are categorized as non-editable. Given that the package names of system widgets and soft keyboard widgets do not change in different apps on the same device, RIDA identifies them based on their `package` attributes in the UI hierarchy. In the cross-app replaying phase, widgets in distinct categories are not expected to be matched with each other.

3) Descriptors Extraction: RIDA obtains semantic descriptors from three candidate attributes of the interacted widgets in their UI hierarchies: `text`, `content-desc`, and `resource-id`. These attributes are also considered by previous test reuse approaches [13], [14]. Some clickable layout widgets, e.g., widgets of the class `ViewGroup` [36] and its subclasses, contain textual descriptions in the `text` attributes of their descendants in the UI hierarchies. For these widgets, RIDA also uses the `text` values of their descendants as the descriptors. To normalize the extracted text values, RIDA performs tokenization, lemmatization (reduce the words to their base forms), and removes common widget type descriptors, such as "button" and "fab" (short for Floating Action Button).

When extracting descriptors, all three candidate attributes can be missing. According to [24], more than 77% of apps contain at least one widget that lacks `content-desc`. Furthermore, we crawled the top 500 apps in each of the 32 app categories on Google Play [37] and used Apktool [38] to obtain their front-end XML files. We identified about 16.4 million widgets. Among them, about 4.2 million widgets (25.6%) lack all three attributes. To avoid empty descriptors in such cases, RIDA utilizes LABELDROID [24] to generate a widget icon's image caption and uses it as the semantic descriptor as a fallback choice. Similarly, these image captions will be normalized following the aforementioned process.

After this phase, each recorded event e will be described as a 3-tuple $e = \langle et, wt, D \rangle$, where et denotes one of the three event types (§III-C1) of this event, wt stands for the widget's type (§III-C2), and D is a set of normalized semantic descriptors associated with the interacted widget, which are extracted from its textual attributes or image caption (§III-C3).

D. Cross-App Replaying

The most important task in the cross-app replaying phase is to match the semantic descriptors of widgets in both the source and target apps and identify appropriate events to replay on the target app. This semantic matching process is described as follows. It takes semantic descriptor set D_s of a source event[1] as input. On the target app's screen, RIDA retrieves a series of semantic descriptor sets $\mathcal{D} = \{D_1, D_2, ..., D_n\}$ from all the n widgets. By comparing the similarities between D_s and every $D' \in \mathcal{D}$, we find the most similar one D_t, and its corresponding widget will be matched. RIDA uses the WMD model [27] to compute the distance between a pair of semantic descriptors. When finding the distance between two semantic descriptor sets D_s and D_t, RIDA takes the mean distance of every pair of semantic descriptors $\langle d_s, d_t \rangle$, where $d_s \in D_s$ and $d_t \in D_t$.

The effectiveness of semantic matching can be drastically affected by common words in the semantic descriptors, which are mostly short texts. For example, by applying the WMD pre-trained model, the semantic distance between "add task" and "delete task" is 0.59, while the distance between "add task" and "create todo" is 1.17. Although the texts in the former pair have a shorter distance, the texts in the latter are actually closer in their meanings (in fact, the former pair have opposite meanings). To overcome this problem, before semantic descriptors are fed into the WMD model, RIDA removes their common words based on the results of part-of-speech (POS) tagging. POS tagging is a process of assigning grammatical tags, such as nouns, verbs, and adjectives, to the words in a sentence according to their context. A common word x will be removed from a pair of semantic descriptors $\langle d_s, d_t \rangle$ before computing their semantic distance if all of the three conditions are satisfied:

1) x has the same grammatical tag in both d_s and d_t
2) d_s contains a non-common word x_s and d_t contains a non-common word x_t
3) x_s and x_t are different but have the same tag

The conditions ensure that for two semantic descriptors, their common word x belongs to the same part-of-speech, while both of them contain non-common words in some other part-of-speech. For example, when finding the semantic distance of two phrases "create note" and "delete note", the identical noun "note" will be discarded, and the similarity between "create" and "delete" will be considered instead because both of them have the same noun but different verbs. After that, their semantic distance is greatly increased. Thus RIDA can achieve better widget matching performance.

As we pointed out before, the source events and the target events may not always be a one-to-one correspondence. Instead, the same usage scenario may require different lengths of interaction steps in different apps. Previous test reuse approaches mostly utilize apps' window transition graphs to match widgets [13], [14]. However, building such graphs can be time-consuming, especially when the app has a large size. Besides, it is also hard to ensure the quality of the graphs as building them typically requires static program analysis, which is often inaccurate. To realize cross-app widget matching, we design a lightweight on-the-fly search algorithm.

Algorithm 1 shows how RIDA performs event matching via bidirectional search. Its main loop has four steps:

Step 1 (lines 3-7): RIDA first attempts to match the current event $E[i]$. In line 3, RIDA finds all interactable widgets of the same type as the widget of $E[i]$. A widget is considered interactable if its `clickable` attribute is `true` on the current

[1] "Source events" refer to the user events recorded in the source app, and "target events" refer to those that will be replayed on the target app. "Source widgets" and "target widgets" are their corresponding widgets, respectively.

Algorithm 1 Cross-App Replaying

Input List of source events E of length N, target app A, semantic matching threshold τ, forward-searching steps m

```
1:  i ← 0
2:  while i < N do
3:      W ← interactable widgets of type E[i].wt on A
4:      w_t ← GETMATCH(E[i].D, W, τ)
5:      if w_t ≠ null then
6:          perform E[i].et on w_t
7:          i ← i + 1
8:      else if i < N − 1 then
9:          j ← 1
10:         while i + j < N − 1 ∧ j ≤ m do
11:             W′ ← interactable widgets of type E[i + j].wt on A
12:             w_t ← GETMATCH(E[i + j].D, W′, τ)
13:             if w_t ≠ null then
14:                 perform E[i + j].et on w_t
15:                 i ← i + j + 1
16:                 break
17:             end if
18:             j ← j + 1
19:         end while
20:     end if
21:     if w_t = null ∧ i > 0 then
22:         W′ ← interactable widgets of type E[i − 1].wt on A
23:         w_t ← GETMATCH(E[i − 1].D, W′, τ)
24:         if w_t ≠ null then
25:             perform E[i − 1].et on w_t
26:         else
27:             w_t ← GETMATCH(E[i].D, W, ∞)
28:             perform E[i].et on w_t
29:             i ← i + 1
30:         end if
31:     end if
32: end while
```

UI hierarchy. Then, it tries to obtain the target widget for $E[i]$. RIDA identifies target widget w_t with the GETMATCH function. The GETMATCH function returns the widget w_t whose descriptor set has the shortest semantic distance with the source widget's descriptors $E[i].D$ among the candidate widgets W, and the shortest distance should be less than a given threshold τ. If a target widget w_t is identified, RIDA will replay $E[i]$ on it and continue with the next iteration.

Step 2 (lines 8-20): If no widget was matched in the first step, RIDA will search forward in E and try to match one of the subsequent m events (line 10), where m is a pre-defined parameter of forward-searching steps. If a subsequent event $E[i+j]$ is matched, RIDA will replay $E[i+j]$ on the target widget and skip other events that precede it (lines 14-15). This step handles the situation where multiple consecutive source events are mapped to one target event.

Consider an example in Figure 3, where we record an "Add note" use case on the source app BlackNote [39] and replay it on the target app Notepad [40]. In BlackNote, adding a new note requires three actions: clicking the "add" button ①, clicking the button with the label "note" ②, and entering note title to the edit title area ③. However, in Notepad, users can input text to the title input box ⑤ directly after a single click on the "add note" button ④. To migrate the source event sequence, by Step 1, RIDA matches source widget ① to "add note" button ④ in Notepad. Then in the next iteration,

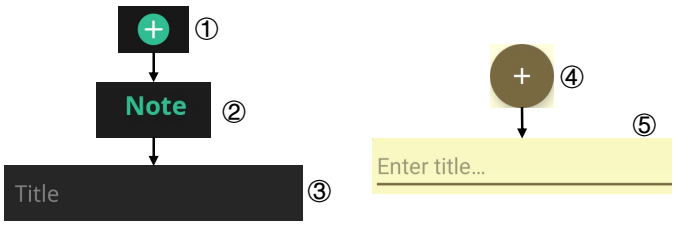


Fig. 3. "Add note" in BlackNote (left) and Notepad (right)

Step 1 fails to identify any matching widget for widget ②. However, by Step 2, RIDA searches forward and successfully finds the target widget ⑤ for the third source event, i.e., the title entering event. Further, we can easily extend this example to a many-to-one event mapping situation, where there are more than two source widgets preceding ③.

Step 3 (lines 21-25): If Step 2 fails, RIDA will search backward and try to match the last event $E[i-1]$ (lines 22-23). If it was matched, RIDA replays the event on the target widget (line 25). After that, the index i will not increase, which means RIDA will attempt to match the current event $E[i]$ again in the next iteration. Step 3 works when one event in the source app is mapped to multiple consecutive events in the target app.

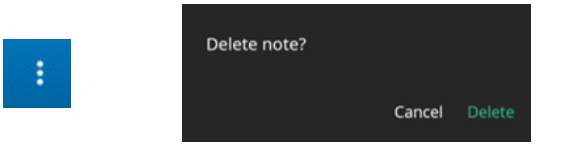
Take Figure 3 as an example again. This time, Notepad is the source app, and BlackNote is the target app. In Step 1 of the first iteration, RIDA will match the source widget ④ to the "add" button ①. In the next iteration, RIDA cannot identify a target widget for the source widget ⑤ on the current GUI by neither Step 1 nor 2. However, by Step 3, RIDA will search backward and match the first event again to the "note" button ②. The intuition is, for one-to-many event mapping, the source event often contains descriptions related to all the mapped target widgets. For instance, ④ contains a descriptor (e.g. "add note") that is semantically similar to the descriptors of ① (e.g. "add") and ② (e.g. "note"). Similarly, this example can be extended to a one-to-many mapping situation.

Step 4 (lines 26-30): If none of the widgets is matched after the above-mentioned attempts, RIDA will fallback and try to match the current event $E[i]$ again. However, this time, the widget with the shortest semantic distance will be directly returned without considering the threshold (line 27). The cross-app replaying process terminates when the source event sequence has been exhaustively processed.

To further improve the effectiveness of widget matching, we integrate two heuristic strategies into the cross-app replaying phase, which correspond to two common interaction patterns on the Android platforms. Note that, in the self-replaying phase, these two types of interactions will not be saved since their implementations may vary in different apps.

First, in the target app, a user sometimes needs to first press a "more options" button to open an options menu, as shown in Table II, to interact with the target widget from the newly appeared options. However, in the source app, the corresponding widget may exist outside the menu. In order to locate these hidden widgets, RIDA first identifies such a menu

TABLE II
"MORE OPTIONS" BUTTON AND CONFIRMATION DIALOG



More Options | Confirmation Dialog

button if its descriptor contains "more options" (e.g., "show more options" and "more options button icon"). If a "more options" button is located on the current UI hierarchy when finding interactable buttons (lines 3, 11, 22), RIDA will click it and add the newly appeared widgets to the candidate list. If the target widget is found behind the "more options" button, RIDA will perform the event on the target widget; otherwise, RIDA will press back and continue matching.

Second, in some scenarios, after users perform an action, a confirmation dialog will pop up, and the users should click the positive button (e.g., "Delete" in Table II) to confirm their previous action. This dialog may not be necessary for a usage scenario. Therefore, every time after RIDA replays an event, it will check if a confirmation dialog appears and click the positive button. RIDA considers a newly appeared layout widget as a confirmation dialog if it contains at most three buttons and their descriptors contain words like "Cancel", "No", or "Yes". Once a dialog is identified, RIDA will click its positive button before replaying the subsequent events.

IV. EVALUATION

A. Implementation Details

RIDA uses UIAutomator2 [7] to perform UI events, obtain the UI hierarchies, and capture screenshots. It applies the standard WMD pre-trained model [41] for semantic analysis of texts. According to [16], a threshold value of 0.65 yields the best trade-off for applying the WMD model. Therefore, we use this value as the threshold for semantic matching. For Algorithm 1, we set the forward-searching step m to be 1 by default because one-to-one and two-to-one cross-app event mappings are frequently observed.

B. Experiment Setup

We conducted all the experiments on a Pixel 3 device running a stock Android 10 system. To collect subject apps, we selected five app categories following the previous test reuse and record-and-replay works on Android [21], [23]. For each category, we searched its name on Google Play and downloaded the first five apps from the query results as our experiment subjects. The subject apps, as well as their exact versions and download numbers, are listed in Table III.

The experiments aimed to answer the following research questions regarding RIDA's effectiveness and usability:

- **RQ1:** How effective is RIDA in performing cross-app record-and-replay for common usage scenarios of real-world Android apps?
- **RQ2:** Can semantic matching and image captioning help improve the effectiveness of cross-app record-and-replay?
- **RQ3:** How does the semantic matching algorithm of RIDA compare to the state-of-the-art approach?
- **RQ4:** How effective is RIDA in performing cross-app record-and-replay for non-designated user interactions?

TABLE III
SUBJECT APPS IN THE EXPERIMENTS

Category	App Name	Version	Downloads
Tasks	Google Tasks [42]	4.23	10M+
	To Do [43]	12.7	10M+
	Tasks [44]	2.7.1	1M+
	Tasks.org [45]	1.8.7	100K+
	Todoist [46]	v10362	10M+
Email	Gmail [47]	2022.08.21	10B+
	Outlook [48]	4.2204.5	500M+
	Email [49]	1.30.0	5M+
	Yahoo Mail [50]	6.59.0	100M+
	Mail.ru [51]	14.31	100M+
Browser	Brave [52]	1.41.100	100M+
	Edge [53]	103.0.1264	10M+
	DuckDuckGo [54]	5.130.0	10M+
	Firefox [55]	70.3.3653	100M+
	UC Browser [56]	13.4.0.1306	1B+
Notes	Notepad [40]	1.21.2	10M+
	BlackNote [39]	2.2.1	5M+
	ColorNote [57]	4.4.0	100M+
	BasicNote [58]	1.1.9	5M+
	Google Keep [59]	3.3.7	1B+
Calculator	Google Calculator [60]	8.2	1B+
	Simple Calculator [61]	1.11.1	100K+
	Calculator Plus [62]	4.2.0	50M+
	HiPER [63]	12.3.12	10M+
	All-In-One Calculator [64]	5.9.1	5M+

To answer the above research questions, we designed two types of experiments:

1) Controlled Experiment: This experiment aims to answer **RQ1**, **RQ2** and **RQ3**. We compared RIDA against a state-of-the-art semantic matching algorithm and three RIDA's variants. They correspond to the following four configurations:

RIDA with SEMFINDER (RIDA w/ SF). SEMFINDER is the state-of-the-art semantic matching algorithm for Android test reuse [23]. SEMFINDER concatenates all the candidate text attributes into one sentence and removes duplicate words, and then compares two sentences by word embedding techniques. Given that an attribute value is usually a phrase or sentence, e.g., "search email", the way SEMFINDER processes them can break their original structures, making syntactic analysis inapplicable. In contrast, RIDA's semantic matching computes the average semantic similarity score between each pair of descriptors and utilizes POS tagging to assist matching. In this configuration, we replaced the semantic matching algorithm of RIDA with SEMFINDER in the cross-app replaying phase.

RIDA without Image Captioning (RIDA w/o IC). In this configuration, RIDA does not use the image captions generated by LABELDROID. In such cases, widgets with empty descriptor sets will have an infinite semantic distance from others. We use these variants to evaluate the impact of image captioning on the effectiveness of RIDA.

TABLE IV
APP CATEGORIES AND CORRESPONDING USAGE SCENARIOS

Category	Usage Scenario	Steps
Tasks	Add new task	1. Click “add task” button 2. Input task title 3. Click “save” button
	Delete task	1. Click/long-click a task 2. Click “delete” button
	Edit task	1. Click a task 2. Clear old task title 3. Input new task title
Email	Send email	1. Click “compose” button 2. Input recipient address 3. Press enter 4. Input title 5. Input mail content 6. Click “send” button
	Search email	1. Click “search” button 2. Input keywords 3. Press enter
	Save draft	1. Click “compose” button 2. Input mail content 3. Click ”back” button 4. Click ”save” button
Browser	Access website by URL	1. Input URL 2. Press enter
	Add new tab	1. Click “switch tab” button 2. Click “new tab” button
	Back button	1. Input URL 2. Press enter 3. Input another URL 4. Press enter 5. Press back
Notes	Add note	1. Click “add note” button 2. Input note title 3. Input note content 4. Press back
	Delete note	1. Click/long-click a note 2. Click “delete” button
	Search note	1. Click “search” button 2. Input keywords 3. Press enter
Calculator	Calculate 1+2	1. Click “1” button 2. Click “+” button 3. Click “2” button 4. Click “=” button
	Clear input	1. Click “6” button 2. Click “-” button 3. Click “3” button 4. Click “clear” button
	Delete number	1. Click “0” button 2. Click “delete” button

RIDA without Semantic Matching (RIDA w/o SM). In this variant, RIDA's semantic matching is replaced by string equality checking, which means two widgets will be matched if they contain identical semantic descriptors in their sets D.

RIDA without Image Captioning and Semantic Matching (RIDA w/o IC & SM). In this configuration, we do not use image captions as widget descriptors and replace semantic matching of RIDA with string equality checking. We use this variant to investigate how the combination of image captioning and semantic matching affects the performance of RIDA.

We designed three common usage scenarios for each app category following the previous work [13] or the subject apps' descriptions. We also manually examined these scenarios to ensure that they can be performed on all subject apps. The steps for each usage scenario are listed in Table IV. Given that a step may correspond to more than one event in some apps, the actual event lengths are between 2 and 7.

In a test, we chose one app as the source app to record a designated usage scenario and then used another app in the same category as the target app for replaying the recorded events. Thus there are 5 (#source app) $\times$ 4 (#target app) $\times$ 15 (#scenario) = 300 tests. Each test is repeated with RIDA and the other four configurations. During the tests, we manually inspected the app states (e.g., views, images) to confirm if an event was successfully replayed. A test ends when all events are successfully replayed or if the replay fails at an intermediate step. Finally, we counted the number of replayed events of each test for each configuration and compared their performances based on the number of **completely replayed tests (#CR)**, **partially replayed tests (#PR)**, and **tests that failed at the first event (#F)**.

2) In-The-Wild Experiment: The controlled experiment has a limited number of usage scenarios. To further evaluate the usability of RIDA, we conducted an in-the-wild experiment to investigate whether RIDA can help replay uncontrolled user interactions with real-world apps (human testers may freely explore a source app's functionalities). For the experiment, we invited three non-Computer Science students to perform 5-10 actions on each subject app in any way. Then, for each test, we used RIDA to replay the recorded event sequence on other subject apps from the same category. Considering that the target apps may not provide certain functionalities of the source app, we dropped the unsupported events before replaying each test. If a source event sequence starts with an unsupported event on a given target app, we simply dropped this test since the following source events are not expected to be applicable from the starting page of the target app. Similar to the controlled experiment, we manually checked if each event is successfully replayed and recorded RIDA's #CR, #PR, and #F for this experiment.

C. Answer to RQ1

As shown by the results in Table V, RIDA can completely replay 164 (55%) and partially replay 105 (35%) out of the 300 tests. In general, RIDA has a higher success rate for shorter scenarios. It even achieves a 100% complete replay for the “Delete note” scenario, which has an average event sequence length of 3. Although RIDA has relatively lower #CR for long scenarios, e.g., “Send email” and “Add note”, it can still partially replay most of the tests. For the long sequence scenario “Back button”, RIDA completely replays 12 out of the 20 tests, while in two short scenarios “Add new task” and “Add new tab”, it completely replays less than a half. The reason for RIDA's poor performance for the two short scenarios is that they involve widgets that lack all candidate attributes. Such a widget can negatively affect eight test results (four tests where the app with the widget is the source app, plus four tests where the app is the target app). Even though RIDA mitigates the issue with image captioning, it can still

TABLE V
CONTROLLED EXPERIMENT RESULTS

Category	Usage Scenario	Avg. #Source Event	RIDA	RIDA w/ SF	RIDA w/o IC	RIDA w/o SM	RIDA w/o SM & IC
Tasks	Add new task	3	5/12/3	4/11/5	3/14/3	4/5/11	4/5/11
	Delete task	3.2	15/5/0	13/4/3	15/5/0	6/14/0	6/14/0
	Edit Task	2.2	19/1/0	16/4/0	19/1/0	20/0/0	20/0/0
Email	Send email	6.2	2/17/1	0/12/8	1/11/8	0/6/14	0/6/14
	Search email	3	15/1/4	14/2/4	15/1/4	4/2/14	4/2/14
	Save draft	3.2	11/9/0	7/9/4	7/9/4	0/6/14	0/6/14
Browser	Access website by URL	2.4	12/4/4	12/2/6	12/4/4	6/0/14	6/0/14
	Add new tab	2	5/9/6	4/10/6	4/10/6	2/12/6	2/12/6
	Back button	5.8	12/4/4	12/1/7	12/4/4	6/0/14	6/0/14
Notes	Add note	4.4	6/10/4	7/7/6	6/10/4	0/1/19	0/1/19
	Delete note	3	20/0/0	16/4/0	20/0/0	12/8/0	12/8/0
	Search note	3.6	15/0/5	16/0/4	15/0/5	6/0/14	6/0/14
Calculator	Calculate 1+2	4	5/15/0	4/16/0	5/15/0	2/13/5	2/13/5
	Clear input	4	4/16/0	4/16/0	4/16/0	0/13/7	0/13/7
	Delete number	2	18/2/0	18/2/0	18/2/0	6/6/8	6/6/8
Total		3.5	164/105/31	147/100/53	156/102/42	74/86/140	74/86/140

[1] The third column is the average length of recorded event sequences for each scenario, which is different from the steps in Table IV because one step can map to multiple user events in different apps.
[2] Each table cell displays three metrics: #CR, #PR, and #F, separated by two slashes.

cause multiple failures when the widget is not image-based or when the image captions are inaccurate. These causes of failure will further be categorized and discussed in §V.

For category-wise performance, RIDA achieves the highest #CR of 41 for the Notes category. With manual analysis, we found that it is because the subject apps in this category are well-maintained. Their widgets' textual attributes can precisely describe the underlying functionalities. Such high-quality semantic descriptors greatly improve the success rate of widget matching between source and target apps. In contrast, for the Calculator category, RIDA has the lowest #CR. RIDA's poor performance for this category is due to the lack of meaningful descriptors in the widgets. Many widgets in the Calculator apps lack textual labels or only contain vague descriptors such as "c" for "clear input" and character "-" for "minus".

D. Answer to RQ2

The results of RIDA's variants are shown in the last three columns of Table V. We compare them with RIDA to see how semantic matching algorithm and image captioning can improve RIDA's effectiveness.

Overall, RIDA achieves the best performance compared with all its variants. RIDA without image captioning performs slightly poorer than RIDA (11 more tests fail to replay at the first step). RIDA without semantic matching is the least effective. Interestingly, the last two variants yield the same results, which means LABELDROID does not generate image captions identical to the existing descriptors and, therefore, cannot improve the effectiveness of string matching.

Compared to the variant without image captioning, RIDA has more complete and partial replays in four scenarios. These scenarios involve interactions with image-based widgets that lack valid textual descriptors, and image captioning helps semantic matching in this situation. As mentioned in §IV-C, one source or target widget that lacks textual descriptors can negatively affect the results. The relatively minor improvement by image captioning is due to the inaccurate or incorrect image captions generated by LABELDROID.

RIDA is more effective than its variant without semantic matching in terms of #CR and #PR. This is because different developers may use different descriptions for widgets with the same functionality. Thus string equality checking cannot help match widgets across apps effectively. An exception occurs for the scenario "Edit task". In that case, the source widget and the expected target widget both contain an identical descriptor. However, RIDA does not guarantee to match such a widget since it may have other descriptors, making the average distance above the threshold. In contrast, string equality checking identifies the target if one descriptor is matched.

In summary, the results show that both semantic matching and image captioning can improve the performance of RIDA in cross-app record-and-replay in terms of #CR and #PR.

E. Answer to RQ3

We compare RIDA without image captioning and RIDA with SEMFINDER to answer RQ3. The reason is that neither of them uses image captions as descriptors. Therefore, their comparison can fairly evaluate RIDA's semantic matching algorithm. Here we refer to the two methods as RIDA′ and SEMFINDER for convenience. Their evaluation results are presented in the sixth and fifth columns of Table V, respectively. RIDA′ achieves nine more #CR and two more #PR than SEMFINDER in total. It has a higher #CR in six scenarios and more #F in only one scenario. Even though most widget descriptors in Calculator apps are single words, the result of RIDA′ and SEMFINDER can still be different due to the difference in how they aggregate similarity scores for two sets of descriptors. The results suggest that RIDA's semantic matching algorithm outperforms SEMFIDNER in cross-app record-and-replay.

F. Answer to RQ4

Table VI shows the results of the in-the-wild experiment. We obtained a total of 267 test results from the three students.

TABLE VI
IN-THE-WILD EXPERIMENT RESULTS

Category	Avg. #Source Event	#CR	#PR	#F	#U
Tasks	3.1	33	13	7	7
Email	3.8	21	23	6	10
Browser	2.9	36	11	9	4
Notes	3.6	23	18	7	12
Calculator	7.9	9	50	1	0
Total	4.3	122	115	30	33

TABLE VII
FAILED CASES

Cause of failure	Controlled	In-the-wild	Total
Inaccurate matching	103	104	207
Incorrect image caption	19	29	48
Empty XML attributes	7	8	15
Incorrect `clickable` attribute	3	2	5
Unrecognized "more options"	4	0	4
Scroll down list	0	2	2

After discarding 33 tests that start with unsupported events (see column #U), the remaining source event sequences have an average length of 4.3. The reason why the first four categories have shorter event lengths is that many subject apps under these categories contain widgets related to app-specific functionalities that are not seen on other apps, and the testers frequently interacted with them. These unsupported events were dropped before we performed cross-app replaying.

RIDA successfully replayed 122 (46%) and partially replayed 115 (43%) of the tests. For the Tasks and Browser categories, RIDA completely replays more than half of the tests. Similar to the results in the controlled experiment, the success rate of RIDA is also highly correlated with the average number of source events. The Browser category, which has the highest success rate, also has the lowest test length, while the Calculator category has a reversed trend. It is worth noting that even though RIDA only achieves a #CR of 9 for the event sequences for Calculator apps due to the long sequence lengths, it partially replays most of the tests, with only one completely failed case.

V. DISCUSSIONS

A. Failed Cases

Table VII shows the number of failed cases and their causes for RIDA in the controlled and in-the-wild experiments, which were verified by us during experiments.

Most failures are caused by the inaccurate matching of target widgets. We observed 103 such failed cases in the controlled experiment and 104 in the in-the-wild experiment. These failures often occur when the original textual attributes are misleading or when incorrect POS tags are generated, etc. In 19 cases of the controlled experiment and 29 cases of the in-the-wild experiment, the image captions generated by LABELDROID are inaccurate or incorrect, leading to replay failures. Incorrect image captions do not improve semantic matching accuracy. They could even trigger unexpected failures if the wrong captions happened to be semantically similar to widgets other than the target ones. For example, the "plus" button in the HiPER calculator [63] is incorrectly labeled as "add contact" by LABELDROID. 15 cases failed due to the empty attributes in the source or target widgets. LABELDROID cannot generate captions for widgets that are not image-based, e.g., editable text areas. As a result, RIDA could not extract their semantic descriptors if all three candidate attributes were empty, and the matching will fail. In five cases, the `clickable` attribute in the UI hierarchy is incorrect, such that unclickable or invisible widgets are marked as clickable. In these cases, non-interactable widgets will be matched if their descriptors match those of the source widget. A typical example is the descriptive text [3] surrounding the actual target widget. There are four cases where RIDA cannot correctly recognize the "more options" button, while the target widget is behind it according to our manual validation. Two failed cases are related to scrollable lists. In these cases, the target widget cannot be seen before we scroll the list down to the appropriate positions.

B. Threats to Validity

One threat to validity is that the apps and usage scenarios we used in the experiment may lack generality. To mitigate this threat, we collected 25 commercial apps from Google Play from five different categories. Our experiment scale is comparable to previous test reuse works [13] [23]. Furthermore, we conducted an in-the-wild experiment, which incorporated additional and uncontrolled usage scenarios produced by human testers. Another threat to validity is the possible mistakes in our implementation and experiment. To mitigate this threat, we have manually inspected all the results and the intermediate files. Moreover, we make our experiment data public for external validation [25].

VI. RELATED WORK

A. Record-And-Replay

There have been many record-and-replay tools for the Android platform. They can be grouped into the following two categories according to the way they record input UI events.

Coordinate-based: RERAN [18] is an early-stage record-and-replay tool on the Android platform. It uses the `getevent` command to capture low-level UI events and performs exact replays of these events. MOSAIC [19] records low-level events and transforms them into a device-independent format, allowing the events to be replayed across different devices by linearly scaling the interacting coordinates according to the target device's screen resolution. VALERA [20] is a stream-oriented record-and-replay approach, it achieves high-accuracy and low-overhead by eliminating the nondeterminism of three system events: sensor input, network, and event schedule. Under such precise timing control, VALERA is able to reproduce event-driven race bugs, which are extremely sensitive to event scheduling. MOBIPLAY [65] captures app inputs from the local device and replays them on the target app running on the server. Such client-server architectural design

allows the testers to record and replay system events without acquiring root privilege from the Android system.

Widget-based: Espresso [66] is the official UI testing framework of Android. It provides a utility Espresso Test Recorder that captures UI events and records widget-based actions with their unique resource IDs. SARA [21] intercepts and records the input events on the application layer via dynamic instrumentation. It can record widget-sensitive events and replay them on devices with different screen resolutions. LIRAT [22] is capable of performing cross-platform record-and-replay. It detects target widgets on different platforms by matching images and page layouts. RANDR [67] achieves cross-device replay with both static and dynamic instrumentation, and it has a special focus on replaying several network protocols and random numbers.

In essence, RIDA is also a widget-based record-and-replay approach. It utilizes the `getevent` command to obtain low-level UI events like many coordinate-based methods. Similar to SARA, RIDA introduces an extra self-replaying phase to categorize input events and extract semantic descriptors, which will be used during cross-app replay. However, different from existing work, RIDA generates image captions and uses word embedding techniques in the replaying phase to detect target widgets on another app with similar functionality. To the best of our knowledge, RIDA is the first tool that achieves cross-app record-and-replay on the Android platform.

B. Test Reuse

Test reuse (a.k.a. "test transfer" or "test migration") is a technique that migrates test scripts across different apps with similar functionalities. A key challenge for test reuse is how to match similar widgets across different apps, i.e., semantic matching. CRAFTDROID [13] and APPTESTMIGRATOR (ATM) [14] are two test reuse tools for Android apps. Both of them use Word2Vec [26] as the word embedding model for semantic matching and analyze the Window Transition Graphs to search for unmatched widgets across different app pages. In contrast, RIDA leverages an on-the-fly searching algorithm to achieve cross-app event mapping, which means RIDA does not need to perform static analysis on the code of each Android Activity and Fragment. Prior to ATM, Behrang and Orso proposed GUITESTMIGRATOR (GTM) [15] for migrating test event sequences. GTM pre-processes texts associated with a pair of given widgets by POS tagging and then calculates their similarity score using the *Wu and Palmer* [68] method. Different from ATM, GTM does not use word embedding and cannot migrate test oracles. ADAPTDROID [16] utilizes the evolutionary algorithm to migrate tests across Android apps. Mariani et al. [23] compared CRAFTDROID and ATM by studying the effectiveness of four components in semantic matching: corpus, word embedding model, event descriptor, and semantic matching algorithm. They did not compare GTM since it was a downgraded version of ATM. They also presented a new semantic matching algorithm SEMFINDER, which outperforms the algorithms of both CRAFTDROID and ATM. Zhao et al. [69] presented a framework FRUITER for evaluating UI test reuse techniques on Android. Unlike [23], FRUITER evaluates test reuse tools as a whole, rather than considering semantic matching algorithms in particular. Both of them do not evaluate the effectiveness of migrating oracles. Recently, Liu et al. [70] present TRASM, which uses adaptive strategies to improve the performance of semantic matching for test reuse on Android Applications.

The key difference between the above test reuse tools and RIDA is that the latter does not depend on the existing test scripts to generate new test cases. Instead, RIDA accepts manual inputs directly from the testers and replays them across apps with similar functionalities. As for end-to-end comparisons between RIDA's cross-app record and replay with other test reuse approaches that are based on existing test scripts, we leave this as our future work.

Test reuse techniques can also be employed for the same app on different application platforms. TESTMIG [71] can migrate GUI tests from iOS to Android. It leverages the sequence transduction technique to guide the event exploration, which explicitly addresses the challenge that users on different mobile platforms may have different habits. MAPIT [72] is capable of performing bidirectional test migration between pairs of Android and iOS apps. Given that app widget appearances are more consistent under this scenario, we believe that RIDA's novel semantic matching algorithm with image captioning technique can also enhance such cross-platform test reuse by improving widget matching accuracy.

VII. CONCLUSION AND FUTURE WORK

This paper proposes a cross-app record-and-replay approach RIDA. RIDA employs an on-the-fly search algorithm to match widgets across apps. It also applies image captioning to generate semantic descriptors when text attributes are unavailable for a given widget. We evaluate RIDA using controlled and in-the-wild experiments. The results show that RIDA can effectively perform cross-app record-and-replay in both scenarios. Also, RIDA's semantic matching algorithm outperforms the state-of-the-art approach and other baseline methods.

One future work is that we can extend RIDA to realize cross-platform record-and-replay, for instance, recording tests in Web applications and replaying them on mobile apps. In addition, RIDA's word-matching algorithm is rather simple. It takes advantage of the syntactic information between descriptors to mitigate the effect of common words on matching accuracy. More advanced techniques can be applied to further improve the RIDA's matching accuracy.

ACKNOWLEDGMENT

The authors would like to thank the ICST 2023 reviewers for their comments and suggestions. This work is supported by the Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562) and the National Natural Science Foundation of China (Grant Nos. 61932021, 61802164), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001).

References

[1] Statista. Android - statistics & facts. Accessed October 8, 2022. [Online]. Available: https://www.statista.com/topics/876/android

[2] Appbrain. Accessed October 9, 2022. [Online]. Available: https://www.appbrain.com/

[3] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao, "Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 257–268.

[4] S. L. Lim, P. J. Bentley, N. Kanakam, F. Ishikawa, and S. Honiden, "Investigating country differences in mobile app user behavior and challenges for software engineering," in *IEEE Transactions on Software Engineering*, vol. 41, no. 1. IEEE, 2014, pp. 40–64.

[5] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu, "Guileak: Tracing privacy policy claims on user input data for android applications," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 37–47.

[6] G. Nudelman, *Android design patterns: interaction design solutions for developers*. John Wiley & Sons, 2013.

[7] openatx. uiautomator2. Accessed October 1, 2022. [Online]. Available: https://github.com/openatx/uiautomator2

[8] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 269–282.

[9] T. Xu, M. Pan, Y. Pei, G. Li, X. Zeng, T. Zhang, Y. Deng, and X. Li, "Guider: Gui structure and vision co-guided test script repair for android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ESEC/FSE)*, 2021, pp. 191–203.

[10] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "Gui-guided test script repair for mobile apps," in *IEEE Transactions on Software Engineering*, vol. 48, no. 3. IEEE, 2022, pp. 910–929.

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

[12] J.-W. Lin and S. Malek, "Gui test transfer from web to android," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 1–11.

[13] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 42–53.

[14] F. Behrang and A. Orso, "Test migration between mobile apps with similar functionality," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[15] B. Farnaz and O. Alessandro, "Test migration for efficient large-scale assessment of mobile app coding assignments," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 164–175.

[16] L. Mariani, M. Pezzè, V. Terragni, and D. Zuddas, "An evolutionary approach to adapt tests across mobile apps," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2021, pp. 70–79.

[17] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 613–622.

[18] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing-and touch-sensitive record and replay for android," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 72–81.

[19] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "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, 2015, pp. 215–224.

[20] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015, pp. 349–366.

[21] J. Guo, S. Li, J.-G. Lou, Z. Yang, and T. Liu, "Sara: self-replay augmented record and replay for android in industrial cases," in *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis (ISSTA)*. ACM, 2019, pp. 90–100.

[22] S. Yu, C. Fang, Y. Yun, and Y. Feng, "Layout and image recognition driving cross-platform automated mobile testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1561–1571.

[23] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching of gui events for test reuse: are we there yet?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 177–190.

[24] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang, "Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 322–334.

[25] RIDA. [Online]. Available: https://doi.org/10.5281/zenodo.7234185

[26] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space." arXiv, 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[27] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *International conference on machine learning*. JMLR.org, 2015, pp. 957–966.

[28] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar *et al.*, "Universal sentence encoder," 2018. [Online]. Available: https://arxiv.org/abs/1803.11175

[29] M. Z. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga, "A comprehensive survey of deep learning for image captioning," in *ACM Computing Surveys*, vol. 51, no. 6. ACM, 2019, pp. 1–36.

[30] F. Mehralian, N. Salehnamadi, and S. Malek, "Data-driven accessibility repair revisited: On the effectiveness of generating labels for icons in android apps," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2021, pp. 107–118.

[31] Z. Long, G. Wu, X. Chen, W. Chen, and J. Wei, "Webrr: self-replay enhanced robust record/replay for web application testing," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 1498–1508.

[32] Getevent. Accessed October 1, 2022. [Online]. Available: https://source.android.com/docs/core/interaction/input/getevent

[33] Gesturedetector. Accessed October 1, 2022. [Online]. Available: https://developer.android.com/reference/android/view/GestureDetector

[34] Viewconfiguration. Accessed October 1, 2022. [Online]. Available: https://developer.android.com/reference/android/view/ViewConfiguration

[35] AndroidDeveloper. Edittext. Accessed October 1, 2022. [Online]. Available: https://developer.android.com/reference/android/widget/EditText

[36] Viewgroup. Accessed October 1, 2022. [Online]. Available: https://developer.android.com/reference/android/view/ViewGroup

[37] Google. Google play. Accessed October 1, 2022. [Online]. Available: https://play.google.com/

[38] Apktool. Accessed October 1, 2022. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[39] N. Notepad. Blacknote. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=notepad.note.notas.notes.notizen

[40] atomczak. Notepad. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.atomczak.notepat

[41] Google. word2vec. Accessed October 1, 2022. [Online]. Available: https://code.google.com/archive/p/word2vec/

[42] G. LLC. Google tasks. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.google.android.apps.tasks

[43] M. Corporation. Microsoft to do: Lists & tasks. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.microsoft.todos

[44] P. B. Limited. Tasks: to do list & tasks. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.tasks.android

[45] Tasks.org. Tasks.org. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=org.tasks

[46] P. B. Limited. Tasks: to do list & tasks. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.todoist

[47] G. LLC. Gmail. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.google.android.gm

[48] M. Corporation. Outlook. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.microsoft.office.outlook

[49] E. Software. Email. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.easilydo.mail

[50] Yahoo. Yahoo mail. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.yahoo.mobile.client.android.mail

[51] M. Group. Mail.ru. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=ru.mail.mailapp

[52] B. Software. Brave. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.brave.browser

[53] M. Corporation. Edge. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.microsoft.emmx

[54] DuckDuckGo. Duckduckgo. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.duckduckgo.mobile.android

[55] Mozilla. Firefox fast & private browser. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=org.mozilla.firefox

[56] U. S. P. Ltd. Uc browser-safe, fast, private. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.UCMobile.intl

[57] Notes. Colornote. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.socialnmobile.dictapps.notepad.color.note

[58] N. Notepad. Basicnote. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=notizen.basic.notes.notas.note.notepad

[59] G. LLC. Google keep. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.google.android.keep

[60] ——. Calculator. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.google.android.calculator

[70] S. Liu, Y. Zhou, T. Han, and T. Chen, "Test reuse based on adaptive semantic matching across android mobile applications," 2023. [Online]. Available: https://arxiv.org/abs/2301.00530

[61] S. D. Ltd. Simple calculator: Math, units. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.candl.athena

[62] L. Digitalchemy. Calculator plus. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=com.digitalchemy.calculator.freedecimal

[63] H. Lab. Hiper scientific calculator. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=cz.hipercalc

[64] allinonecalculator.com. All-in-one calculator. Accessed September 9, 2022. [Online]. Available: https://play.google.com/store/apps/details?id=all.in.one.calculator

[65] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 571–582.

[66] A. Developer. Espresso test recorder. Accessed October 1, 2022. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/espresso-test-recorder

[67] O. Sahin, A. Aliyeva, H. Mathavan, A. Coskun, and M. Egele, "Randr: Record and replay for android applications via targeted runtime instrumentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 128–138.

[68] Z. Wu and M. Palmer, "Verbs semantics and lexical selection," in *Proceedings of the 32nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 1994, pp. 133–138.

[69] Y. Zhao, J. Chen, A. Sejfia, M. Schmitt Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic, "Fruiter: a framework for evaluating ui test reuse," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1190–1201.

[71] X. Qin, H. Zhong, and X. Wang, "Testmig: Migrating gui test cases from ios to android," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 284–295.

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