Cross-Device Record and Replay for Android Apps

Cong Li
State Key Lab for Novel Software
Technology and Department of
Computer Science and Technology,
Nanjing University, Nanjing, China
congli@smail.nju.edu.cn

Yanyan Jiang
State Key Lab for Novel Software
Technology and Department of
Computer Science and Technology,
Nanjing University, Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Lab for Novel Software
Technology and Department of
Computer Science and Technology,
Nanjing University, Nanjing, China
changxu@nju.edu.cn

ABSTRACT

Cross-device replay for Android apps is challenging because apps have to adapt or even restructure their GUIs responsively upon screen-size or orientation change across devices. As a first exploratory work, this paper demonstrates that cross-device record and replay can be made simple and practical by a one-pass, greedy algorithm by the Rx framework leveraging the least surprise principle in the GUI design. The experimental results of over 1,000 replay settings encouragingly show that our implemented Rx prototype tool effectively solved non-trivial cross-device replay cases beyond any known non-search-based work's scope, and had still competitive capabilities on same-device replay with start-of-the-art techniques.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Application specific development environments.*

KEYWORDS

Android app testing, record and replay

ACM Reference Format:
Cong Li, Yanyan Jiang, and Chang Xu. 2022. Cross-Device Record and Replay for Android Apps. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549083

1 INTRODUCTION

Record and replay is a trending technology [24, 41, 49, 54, 61]. In the context of Android apps, record and replay refers to collecting runtime log and replaying the log on later app runs to "reproduce" a past app execution [5, 6, 10, 12, 20, 21, 23, 25, 26, 30, 38, 44, 48, 53, 55, 56, 65]. Record and replay for Android apps is the foundation of a broad spectrum of testing and debugging technologies: failure reproduction [40, 59, 69], regression testing [11, 31, 34, 42, 43, 63, 68], test case minimization [17], to name a few.

The ability of cross-environment adaptation is preferred for modern software, which is expected to grow with continuous evolution for changable environments [62]. This paper thereby studies a relevant problem about cross-environment adaptation: a variant of traditional record and replay for Android apps, namely *cross-device record and replay* (or R&R$_c$ for abbreviation), where an app execution is replayed on a *distinct* device of different screen size or orientation. R&R$_c$ enables developers to "*record once and replay everywhere*" and benefits various testing and debugging practices. With R&R$_c$, cross-device compatibility test cases only need to be recorded on a single device, and regression test cases can be automatically ported to different hardware platforms.

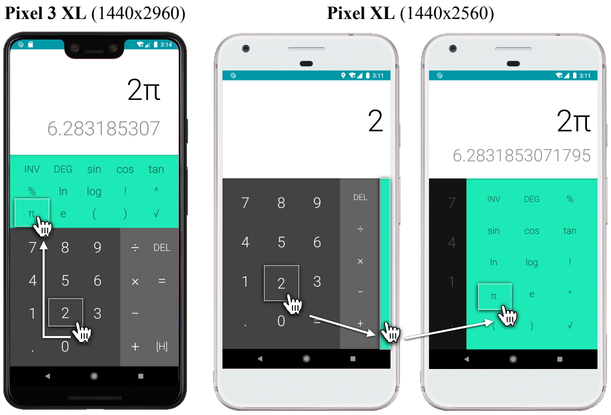

Figure 1: The GUI layout of Google Calculator is responsively restructured to adapt to different screen sizes, posing challenges to cross-device record and replay.

R&R$_c$ is challenging because an app's GUI layout responsively adapts to the screen size and orientation, and the restructured GUI layout may cause exercising the same action on different devices to require quite different event sequences (Figure 1). To our surprise, assuming that each recorded event e's receiver object should present on the replay-time GUI, *no* known replay work [5, 6, 10, 12, 20, 21, 23, 25, 26, 30, 38, 44, 48, 53, 55, 56, 65] is sensitive to GUI restructuring and well supports cross-device record and replay even for the Android's default Calculator app. Although it is theoretically possible to search for a replay (e.g., via test migration [11, 31, 43, 68]), search-based approaches are limited in scalability for industrial-size apps and practical usage scenarios because state-space search incurs costly backtracking and app restarts.

This paper presents the Rx replay framework to develop practical R&R$_c$ implementations, especially for apps with *responsive* GUI restructuring. The framework also works for apps without any GUI restructuring. Specifically, this paper treats an app to have

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11...$15.00
https://doi.org/10.1145/3540250.3549083

"responsive" GUI restructuring if it follows the responsive UI design principle defined officially [18]. The framework leverages the least surprise principle [28, 45, 50] in the field of GUI design that

1. GUI widgets have spatial locality. Specifically, functionally related widgets are spatially adjacent in the GUI layout; and
2. Cross-device GUI adaptation typically obeys a limited set of responsive patterns officially recommended by Android documentation [4, 18].

Accordingly, to replay an event e whose receiver widget is not present on the current GUI layout, the Rx framework follows a natural, greedy procedure without backtracking or app restart by

1. Performing UI segmentation (such that spatially adjacent widgets belong to the same segment) and matching to identify the replay-time correspondence segment of e's receiver object. For example, the "π" button (left) is matched with the green bar (middle) in Figure 1.
2. Tentatively applying a list of revertible responsive actions until e's receiver widget appears (e.g., clicking the green bar). In the case that any reasonable attempt fails, the effects can be reverted by executing its pre-defined inverse action.

This paper further demonstrates that *the Rx framework enables practical cross-device record and replay* even empowered with well-known heuristics for UI segmentation/matching and simple responsive patterns. The experimental results are encouraging that our prototype implementation can correctly complete cross-device replay tasks *beyond any known (non-search- and) event-based replay work's scope* (with an 86.7% successful rate) and still have a competitive same-device replay capability with state-of-the-art same-device only replay techniques.

In summary, this paper's contributions are: (1) the first exploratory work to demonstrate that cross-device record and replay can be made simple and practical, and (2) publicizing the Rx prototype tool and supplementary materials to facilitate future research via

https://sites.google.com/view/rx-framework/home.

The rest of this paper is organized as follows. The cross-device record and replay problem is defined and analyzed in Section 2. The Rx record and replay algorithms (the framework) are described in Section 3. A simple practical realization of the framework (including a prototype implementation) and evaluation results are presented in Sections 4 and 5, respectively. Finally, we discuss related work in Section 6 and conclude the paper in Section 7.

2 PROBLEM AND INSIGHTS

This section presents the problem of cross-device record and replay and our insights to mitigate the challenges. Particularly, we present a formulation of our problem in Section 2.1, analyze the challenges in Section 2.2, and elaborate on our insights in Section 2.3.

2.1 Problem Formulation

Event-Based Record and Replay. This paper's scope is limited to event-based record and replay, the most practical and widely-adopted [30] replay technique by replaying (redoing) a captured sequence of (GUI or system) events on another app execution. Though limited in replaying pixel-precise gestures and system I/O (e.g., network traffics), the lightweight nature of event-based record and replay still makes it a best practice for Android app testing and debugging [11, 31, 42, 63].

In this paper, an event e is an object (a key-value mapping in which *obj*.`field` denotes the value of the key `field`) where:

- e.`type` is the event type, e.g., `ui:click`;
- e.`recv` denotes the event's associated receiver object (an Android *view*[1] for a GUI event, or `null` for a system event);
- e.`params` contains the event's parameters that can not derive from e.`recv`, e.g., an input string for `ui:input` events;
- e.`context` is the event's context object consisting of the timestamp, GUI layout, etc.

To replay a logged event e on a replay device, a typical event-based replay implementation can either deliver the logged low-level event coordinates [10, 12, 21, 25, 44, 65], click the same receiver object by matching widget ID e.`recv.id` [20, 23, 48], or conduct semantic-aware widget matching based on e.`recv.text` and e.`context.UI` [11, 31, 35, 42, 43, 63].

Cross-Device Event-Based Record and Replay. To replay an event sequence $\tau = [e_1, e_2, \cdots, e_{|\tau|}]$ on another device, existing work widely assumes that any e_i's receiver e_i.`recv` should exist at the replay time [5, 6, 10, 12, 20, 21, 23, 25, 26, 38, 44, 48, 53, 55, 56, 65]. Unfortunately, this assumption breaks for real-world R&R$_c$ cases when an app's GUI is responsively restructured to adapt to a device's screen size or orientation. For example, all known replay work failed cross-device replaying the calculation of "2π" in Figure 1. Figure 2 depicts a more complicated (but practical) case that inserting a table in Microsoft Word requires quite different event sequences: $\boxed{1}\rightarrow\boxed{2}\rightarrow\boxed{3}\rightarrow\boxed{4}$ on a phone v.s. $\boxed{1}\rightarrow\boxed{2}$ on a tablet.

2.2 Challenges

Conceptually, R&R$_c$ is a search problem. Given any replay oracle that can determine replay success (e.g., triggering the same crash, manifesting the same GUI changes, or producing a similar log), one can exhaustively try all possible event sequences until an oracle-satisfying one is successfully replayed [11, 31, 40, 42, 43, 59, 63, 69]. However, both the replay oracle and the search are far from trivial in implementing a practical R&R$_c$:

Challenge 1. *(Replay Oracle) There lacks an automatic replay oracle for determining cross-device "replay success" for responsive apps that display different GUI layouts across devices.*

Existing replay oracle either targets same-device replay [12, 21, 26, 40, 44, 59, 65, 69] or optimistically assumes that an app's GUI changes follow limited patterns on distinct devices [5, 6, 10, 20, 23, 25, 38, 48, 53, 55, 56], e.g., resizing a scrollable list. Such replay oracles, insensitive to GUI restructuring, cannot be adopted to real-world R&R$_c$ because an event possibly has no correspondence on the replay device, e.g., $\boxed{1}$ for Microsoft Word in Figure 2.

Challenge 2. *(Search Space) Replaying practical usage scenarios for industrial-size apps yields huge search spaces.*

Although it is theoretically possible to search for a replay (e.g., via test migration [11, 31, 43]), search-based approaches are limited in scalability for industrial-size apps, in which a single GUI

[1] View is the basic GUI widget in Android. Following Android, we use the term "view" to replace "widget" in the following paper.

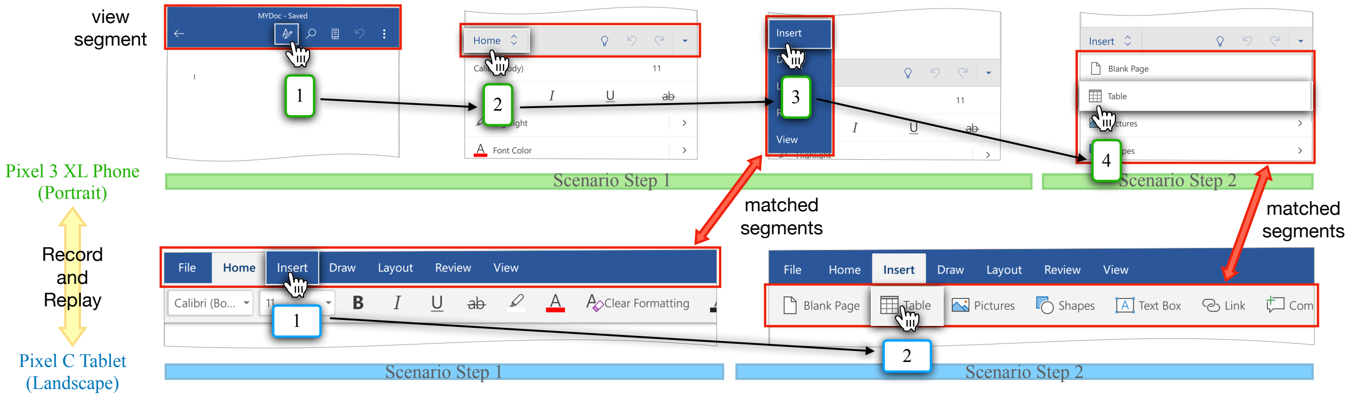

Figure 2: Microsoft Word responsively adapts to screens of different sizes. The event sequences for inserting a table are $\boxed{1}\rightarrow\boxed{2}\rightarrow\boxed{3}\rightarrow\boxed{4}$ and $\boxed{1}\rightarrow\boxed{2}$ on a phone/tablet, respectively. Screenshots are cropped such that only a receiver object's adjacent views are displayed. Key events have a 1-1 correspondence on different devices: $\boxed{3}$ v.s. $\boxed{1}$ and $\boxed{4}$ v.s. $\boxed{2}$. To replay $\boxed{1}\rightarrow\boxed{2}$ on a phone, $\boxed{1}$ and $\boxed{2}$ are required to reveal the "insert" button. In the reverse replay direction, $\boxed{1}$ and $\boxed{2}$ have no correspondence on a tablet and thus should be discarded.

layout may contain tens or even hundreds of clickable views. The attempt to even replay a single event (e.g., $\boxed{1}$ for Microsoft Word) may produce a sub-search-space of millions of event sequences. Efficient implementation of the search is also challenging because backtracking often requires costly app restarts.

2.3 Observations and Insights

Structure of an Event Sequence. An execution to be replayed represents a meaningful *usage scenario*. Natural observation is that all events in a usage scenario are not created equal: there are *key events* indicating a functionally critical action to be performed, while the others are *triggering events* for exposing key events' receiver objects[2]. Consequently, an event sequence can be decomposed into a series of *scenario steps*, each of which ends with a single key event for completing the action preceded by a prefix of triggering events. For example, to insert a table in Microsoft Word (the usage scenario in Figure 2), there are two scenario steps: (1) activate the "Insert" tab, and (2) click the "Table" button. The key events for them are $\boxed{3}$ and $\boxed{4}$ for the phone (or $\boxed{1}$ and $\boxed{2}$ for the tablet), respectively.

Replay Oracle. Despite that not all event receivers have correspondences on another replay device, *key events should maintain a 1-1 mapping between the record and replay devices* since a key event denotes a purposed action in a scenario step and should be executed regardless of device. Our replay oracle thus can be defined as *the in-order replay of all key events*.

Nevertheless, the definition of key/triggering events of a usage scenario is subjective to the concerned developer or app user. For example, a developer may consider $\boxed{1}$ to be the triggering event for $\boxed{2}$ while another may consider $\boxed{1}$ and $\boxed{2}$ to be different scenario steps. Pragmatically, *maximizing the successfully replayed events* (instead of searching for key events) is a reasonably well automatic replay oracle. This is because replaying a triggering event (whose receiver object exists on both devices) is not expected to break the replay procedure.

Search Space. Instead of an enumerative search, we *greedily maximize* the number of events to be replayed by optimistically assuming that each event $e \in \tau$ is a key event and attempting small GUI perturbations to expose e.recv. If the attempt to replay e fails, e should be a triggering event (on the replay device) and we should proceed the replay with e being dropped. The validity of such a surprisingly simple greedy algorithm is threefold:

First, well-designed apps very likely follow the least surprise principle [28, 45, 50] with *the spatial locality of views*, i.e., functionally related views are positioned in a visually adjacent block (namely, a view *segment* in this paper). Therefore, even if a key event e's receiver (e.recv) is not present on the replay device, a human should have no obstacle in identifying e.recv's corresponding view segment on the replay device given the record-time GUI layout. Red boxes in Figure 2 denote our identified segments.

Second, view segments would pragmatically obey limited *responsive patterns*, a small set of officially recommended UI design rules (Material Design [4, 18]) for adapting view segments to different sizes. Each responsive pattern also defines what *(responsive) actions* a human should perform when interacting with that segment. To reveal e.recv on the replay device, one can attempt to perform any responsive action to e.recv's corresponding view segment. For example, ListView is associated with the scrolling action, which is useful in finding e.recv on a smaller device. Another example is that $\boxed{1}$ clicks the Android's official "More Options" button (with a customized icon by Microsoft Word). This is a natural responsive action for revealing a menu for subsequent actions. Focusing only on responsive actions is the key insight to avoid an exhaustive search.

Finally, even if applying a responsive action (e.g., calling out a menu, scrolling a view, etc.) fails to reveal e.recv, its effects can easily be reverted by performing its inverse action (e.g., closing a menu or scrolling in the reverse direction): every responsive action

[2] Such an observation has been exploited in other Android-related tasks, e.g., benchmarking test case reuse [68].

Algorithm 1: The Rx replay framework. Underlined functions are implementation-customizable.

```
   Input: An app $A$ to be replayed on device $\hat{D}$
1  Function RxReplay($\tau$)
2      for each event $e \in \tau$ do
3          $\hat{r} \leftarrow$ empty sequence;
4          repeat
5              $\hat{S} \leftarrow$ FindSegment($e$);
6              if $e$.recv $\in \hat{S}$ then // the receiver object is revealed
7                  send $e$ to $\hat{D}$; break; // replay the next event
8              $\langle \hat{e}, \hat{e}^{-1} \rangle \leftarrow$ FindRespAction($e, \hat{S}$);
9              if $\hat{e} = \perp$ then // $e$ should be a triggering event
10                 send $\hat{r}$ to $\hat{D}$; break; // revert side effects
11             send $\hat{e}$ to $\hat{D}$;   $r \leftarrow [\hat{e}^{-1}] :: \hat{r}$;
12         until TRUE;
13 Function FindSegment($e$)
14     $\mathcal{S} \leftarrow$ SegmentUI($e$.context.UI);
15     $\hat{\mathcal{S}} \leftarrow$ SegmentUI($\hat{D}$.UI);
16     $M \leftarrow$ MaxweightBipartMatch($\mathcal{S} \cup \hat{\mathcal{S}}, W$) where
         $W_{S,\hat{S}}$ = Similarity($S, \hat{S}$) for $S \in \mathcal{S}, \hat{S} \in \hat{\mathcal{S}}$;
17     return $\hat{S} \in \hat{\mathcal{S}}$ where $e$.recv $\in S$ and $(S, \hat{S}) \in M$;
18 Function FindRespAction($e, \hat{S}$)
19     for $P \in$ RespPatterns do
20         $\langle \hat{e}, \hat{e}^{-1} \rangle \leftarrow P(e, \hat{S})$; // Applying a pattern returns a pair of
             events $\langle \hat{e}, \hat{e}^{-1} \rangle$ such that $\hat{e}^{-1}$ reverts $\hat{e}$'s side effects
21         if $\hat{e} \neq \perp$ then return $\langle \hat{e}, \hat{e}^{-1} \rangle$ ;
22     return $\langle \perp, \perp \rangle$;
```

should be reversible in a well-designed UI. If trying all responsive actions still cannot reveal e.recv, e is considered a triggering event on the replay device and thus discarded, yielding our one-pass greedy replay algorithm without costly backtracking or app restart.

In Figure 2, when replaying $\boxed{1}\rightarrow\boxed{2}\rightarrow\boxed{3}\rightarrow\boxed{4}$ on a tablet, $\boxed{1}$ and $\boxed{2}$ are dropped because all attempts to manifest them fail, while $\boxed{3}$ and $\boxed{4}$ are successfully replayed by $\boxed{1}$ and $\boxed{2}$, respectively. When replaying $\boxed{1}$ on a phone, consecutively applying two responsive actions (further explained in Section 4.1) reveals $\boxed{1}$'s receiver object (thus $\boxed{3}$ can be replayed): calling out a menu ($\boxed{1}$) ⟶ revealing a sibling tab ($\boxed{2}$), and $\boxed{2}$ is directly replayed by $\boxed{4}$.

3 THE RX FRAMEWORK

The Greedy Replay Framework. Following the previous analyses, given a recorded event trace τ on device D, the Rx replay framework (Algorithm 1) is an online algorithm that can handle a stream of events without backtracking or app restarts, as contrast to prior work [11, 31, 40, 42, 43, 59, 63, 69]. For each event e to be replayed, the algorithm tries to reveal e.recv's correspondence on the replay device $\hat{D}$ and replay e[3]. Upon failure (Line 9), the algorithm reverts all responsive actions and proceeds with e being discarded.

Algorithm 2: Spatial-locality guided UI segmentation.

```
1 Function SegmentUI($S$)
2     $\mathcal{P} \leftarrow \{\langle S_\ell, S_r \rangle \mid (S_\ell \cup S_r =$
        $S) \wedge$ there exists a horizontal/vertical line separating $S_\ell, S_r\}$;
3     $\langle S_\ell^*, S_r^* \rangle \leftarrow \underset{\langle S_\ell, S_r \rangle \in \mathcal{P}}{\arg\max}$ Naturalness($S_\ell, S_r$);
4     if Naturalness($S_\ell^*, S_r^*$) < THRESHOLD then
5         return $\{S\}$; // return current segment
6     else
7         return SegmentUI($S_\ell^*$) $\cup$ SegmentUI($S_r^*$);
```

Considering that both spatial locality and responsive patterns are design principles from a human perspective, Rx as an extensible framework leaves the three human-related components free customizable (which are underlined in Algorithm 1):

- SegmentUI(S) for partitioning a given UI (S, a set of views) into disjoint view segments. Views within the same segment should be spatially adjacent and functionally related.
- Similarity($S, \hat{S}$) for measuring the similarity between segment S and $\hat{S}$ in terms of app functionality.
- RespPatterns is a list of supported responsive actions. For each pattern P in the list, $P(e, \hat{S})$ returns an event and its inverse $\langle \hat{e}, \hat{e}^{-1} \rangle$ for applying and reverting P to $\hat{S}$. If P is not applicable to $\hat{S}$, $e = \hat{e} = \perp$.

UI Segmentation and Matching. To find an event's receiver object on the replay device, we first segment both GUIs (e.context.UI and $\hat{D}$.UI) into view segments by invoking SegmentUI (Line 5, Lines 13–17). We argue that any non-surprising GUI design should make e.recv's containing segment S to also appear on $\hat{D}$. Therefore, we attempt to find the global maximum weighted bipartite matching between view segments in the two GUIs and narrow the scope of our replay to $\hat{S}$, S's correspondence on $\hat{D}$ (Lines 16–17). In case that e.recv finds its equivalent counterpart in $\hat{S}$, we directly send e to $\hat{D}$ (Lines 6–7) and complete the replay of e.

Applying Responsive Patterns. Once e.recv has no correspondence in $\hat{S}$, the algorithm applies a sequence of responsive actions to reveal e.recv. Sometimes, multiple patterns in RespPatterns can be applied (e.g., Options and Tranx$^{(1)}$ are applied when replaying $\boxed{1}$ on the phone in Figure 2). We generally recommend that the list of responsive patterns RespPatterns can be sorted by the simplicity (simpler patterns are closer to the list head), such that the algorithm (Lines 19–21) automatically returns the simplest pattern following Occam's razor [13] (Line 21). The insight is that UI design should be simple without surprising a human [28, 45, 50], and simplest patterns should involve the least perturbation of GUI and the least side effects.

The attempt to reveal e.recv may be repeated several times (the loop in Lines 4). Each time a responsive action $\hat{e}$ is performed, its inverse $\hat{e}^{-1}$ is pushed to a stack r (Line 11). If all attempts are failed (Line 9), e should be a triggering event on $\hat{D}$, and events in the stack are popped to revert all side effects of responsive actions (Line 10). The replay proceeds with trying to replay the next event.

Sometimes, the greediness of the algorithm may cause an unintended event e (a triggering event on D, but neither key nor

[3] This paper's visual notation convention is that $\hat{\square}$ denotes $\square$'s replay-time counterpart.

Table 1. List of 13 experience responsive patterns.

Rk	Name	**Precondition** ($r = e.\mathtt{recv} \in S$)	$\langle \hat{e}, \hat{e}^{-1} \rangle$
1	Expand$^{(1)}$	$\exists \hat{v} \in \hat{S}$ s.t. $\hat{v}.\mathtt{text}$ starts with $r.\mathtt{text}$ or $r.\mathtt{text}$ starts with $\hat{v}.\mathtt{text}$	`click` $\hat{v}$, `click back`
2	Expand$^{(2)}$	$\exists \hat{v} \in \mathtt{parent}(\hat{S})$ s.t. $\hat{v}.\mathtt{text}$ starts with $r.\mathtt{text}$ or $r.\mathtt{text}$ starts with $\hat{v}.\mathtt{text}$	`click` $\hat{v}$, `click back`
3	Expand$^{(3)}$	$\exists \hat{v} \in \hat{S}$ s.t. $\hat{v}.\mathtt{desc} = r.\mathtt{text}$ or $r.\mathtt{desc} = \hat{v}.\mathtt{text}$	`click` $\hat{v}$, `click back`
4	Scroll	$r.\mathtt{parent}$ is scrollable and $\hat{v} \in \hat{S}$ s.t. $\hat{v}$ is $r.\mathtt{parent}$’s counterpart	`scroll`⇃↾ $\hat{v}$, `scroll`↿⇂ $\hat{v}$
5	Options	$\exists \hat{v} \in \hat{S}$ s.t. $\hat{v}$ is Android’s official “more options” button	`click` $\hat{v}$, `click` $\perp$
6	Menu	$\exists \hat{v} \in \hat{S}$ s.t. $\hat{v}.\mathtt{desc}$ contains text “close/open navigation drawer”	`click` $\hat{v}$, `click back`
7	Tranx$^{(1)}$	r is a `Tab` and $\exists \hat{v} \in \hat{S}$ s.t. r and $\hat{v}$ are siblings	`click` $\hat{v}$, `click back`
8	Tranx$^{(2)}$	$\exists v \in S. \exists \hat{v} \in \hat{S}$ s.t. v is parent of r, v is a `Tab`, and $\hat{v}$ is v’s counterpart	`click` $\hat{v}$, `click back`
9	Pager$^{(1)}$	$r.\mathtt{parent}$ is a `Pager` and $\exists \hat{v} \in \hat{S}$ s.t. $\hat{v}$ is $r.\mathtt{parent}$’s counterpart	`click` $\hat{v}$, `click back`
10	Pager$^{(2)}$	$\exists \hat{v} \in \hat{S}$ s.t. $\hat{v}.\mathtt{parent}$ is `Pager` and r is a child of $\hat{v}$	`click` $\hat{v}$, `click back`
11	Divide$^{(1)}$	$\exists \hat{S}' \in \hat{D}. \exists \hat{v}' \in \hat{S}'$ s.t. $\hat{S}'$ is a `Nav` UI with a nested `List` $\hat{v}'$, and $\hat{S}$ is a `Frag` UI	`click back`, `click` $\hat{v}^{*2}$
12	Divide$^{(2)}$	$\exists \ell \in S. \exists v \in S. \exists \hat{v} \in \hat{S}$ s.t. v’s parent ℓ is a `List`, v is selected, and $\hat{v}$ is v’s counterpart	`click` $\hat{v}$, `click back`
13	Navigate	$\exists \hat{v} \in \hat{S}$ s.t. $\hat{v}$ is Android’s official “navigation up” button	`click` $\hat{v}$, `click` $\hat{v}^{*2}$

[1] Responsive patterns (rows) are listed in the order of simplicity. A responsive pattern returns $\langle \hat{e}, \hat{e}^{-1} \rangle$ when its precondition is satisfied (otherwise, $\langle \perp, \perp \rangle$ is returned). Patterns 1–13 correspond to expand, reveal, transform, and divide patterns defined in Material Design [18]. Patterns 11–12 also appear in Android’s fragment documentation [4].

[2] $\hat{v}^*$ is v’s most semantically related view on $\hat{D}$. Our implementation selects the view of a maximum `Similarity` with $\hat{S}$.

triggering on $\hat{D}$) to be replayed. Such (triggering) events usually do not cause breaking UI changes, and thus replaying them will not likely result in subsequent replay failures. The greedy nature of the algorithm may also drive the replay to a “dead end” in which no responsive pattern is applicable, and all subsequent events are discarded. As shown in our evaluation (Section 5), such failure cases are mainly due to the app not strictly following responsive patterns.

Discussions. There can be alternatives to implement the interfaces (`SegmentUI`, `Similarity`, and `RespPatterns`) required by the Rx framework. Since all the three functions are classifiers related to computer-human interaction, data-driven approach (e.g., statistical learning) certainly applies.

However, considering that there is no publicly available dataset for the cross-device replay task (collecting data and training classifiers is generally less relevant to the R&R$_\mathrm{c}$ problem) and data-driven approach has its unique challenges (e.g., interpretability, hyper-parameter tuning, and privacy issues for using end-user traces), we demonstrate the power of Rx framework by leveraging well-known heuristics and simple rules for which all designs can be well explained and justified.

4 A PRACTICAL REALIZATION OF RX

We encode Android and UI design domain-specific knowledge as heuristic algorithms and responsive patterns as our proof-of-concept Rx instantiation. Despite being simple, the encouraging evaluation results in Section 5 confirmed the practical merits of the Rx framework.

4.1 `SegmentUI`, `Similarity`, and `RespPatterns`

UI Segmentation. Our `SegmentUI` (Algorithm 2) algorithm creates an “interpretable” UI segmentation following the VIPS algorithm [15]. Intuitively, the algorithm resembles printing out all views in S on a paper and conducting the most “natural” horizontal or vertical cutting off, following the intuition that horizontal/vertical lines are the most natural way for human beings to separate functional blocks. We enumerate all possible cut-off lines (Line 2) and find the partition $\langle S_\ell^*, S_r^* \rangle$ that best bisects unrelated functionalities (Line 3). Such “paper cutting” is recursively conducted on both parts S_ℓ^* and S_r^* (Line 7), until splitting S yields an unnatural bisection of views (Lines 4–5).

The naturalness of a bisection is measured by the function `Naturalness`. Specifically, a natural bisection expects that (1) views in the same part are both spatially and functionally “close” to each other, whereas (2) views in different parts are “far away” from each other. Such close/faraway measurements are captured by a set of functions σ_i ($1 \le i \le n$, the smaller the closer), e.g., $\sigma_{\mathrm{bg}}(v, v')$ regards v and v' as close if they have similar background colors. Given σ_i and their weights w_i, $\mathtt{Naturalness}(S_\ell, S_r)$ measures the bisection’s naturalness by a weighted combination

$$\mathtt{Naturalness}(S_\ell, S_r) = \frac{1}{\sum_{1 \le i \le n} w_i} \sum_{1 \le i \le n} w_i \cdot \Bigg(\sum_{v \in S_\ell} \sum_{v' \in S_r} \sigma_i(v, v') - \sum_{v \in S_r} \sum_{v' \in S_r} \sigma_i(v, v') - \sum_{v \in S_\ell} \sum_{v' \in S_\ell} \sigma_i(v, v') \Bigg)$$

in which for v and v' that are far away from the other, splitting them into S_ℓ and S_r makes positive contributions to the naturalness, while pairing them into S_ℓ or S_r makes negative contributions. We explored 6 functions (i.e., σ_i) who share the same weight (i.e., $\frac{1}{6}$) in our prototype. We tuned `THRESHOLD` by Calculator and MS Outlook (and it generalizes well to other subjects (Table 2) in our experiments). Our supplementary material provides more details on how these functions work.

Similarity Measurement. We measure the similarity between segments S and $\hat{S}$ via the textural information of views. For each view v, we collect textual descriptions from $v.\mathtt{text}$, $v.\mathtt{desc}$, $v.\mathtt{id}$, and $v.\mathtt{hint}$. Collected texts are pre-processed with stop words removed, tokenized into words, and bagged up as a document. Afterward, we calculate the TF-IDF [29, 33, 60] vector u and $\hat{u}$ for S’s and $\hat{S}$’s document, respectively. Finally, $\mathtt{Similarity}(S, \hat{S})$ is measured by the standard cosine similarity of their corresponding vectors:

$$\mathtt{Similarity}(S, \hat{S}) = \frac{u \cdot \hat{u}}{\|u\|_2 \times \|\hat{u}\|_2}.$$

Responsive Patterns. We investigated the Top 100 apps in the Google Play Store [22] and distilled 13 frequently used patterns for gracefully responding to different screen sizes and orientations as listed in Table 1. Each patterns is either defined by Material Design [18] (i.e., *Expand*, *Reveal*, *Divide*, and *Transform*) or explicitly mentioned and recommended by Android documentation [4] for user-friendly UIs. For example, on smaller screens, the Options

Table 2. Information of evaluated apps

App Name	Short	Category	#Install
Calculator	GC	Tools	$10^{8.7+}$
YouTube	YT	Video Pla. & Edi	$10^{9.7+}$
MS Todo	MST	Productivity	$10^{6.7+}$
MS Outlook	MSO	Productivity	10^{8}
MS Word	MSW	Productivity	10^{9}
AdobeReader	AAR	Productivity	$10^{8.7+}$
Firefox	FF	Communication	10^{8}
DoodleMaster	DM	Art & Design	10^{6}
AdobeSparkPost	ASP	Art & Design	10^{7}
Zedge	ZDG	Personalization	10^{8}
CollageMaker	CM	Photography	$10^{7.7+}$
Calm	CA	Health & Fitness	10^{7}
Audible	ADB	Books & Ref.	10^{8}
KingJamesBible	KJB	Books & Ref.	10^{7}
Webtoon	WT	Comics	$10^{7.7+}$
ESPN	ES	Sports	$10^{7.7+}$
Discord	DC	Communication	10^{8}
uDates	UD	Dating	$10^{5.7+}$
Remind	RM	Education	10^{7}
Spotify	SP	Music & Audio	$10^{8.7+}$
Reddit	RD	News & Magazines	$10^{7.7+}$
Summary	First Set: 7 Apps; Second Set: 14 Apps		

pattern hides some function-related and less-important buttons via a "More Options" button; while the Divide pattern divides a layered UI into multiple layers (UIs) [18]. Readers may refer to our supplementary material for illustrative descriptions of these responsive patterns.

4.2 The Rx Prototype Tool

We implement a prototype named Rx, which includes the framework (Section 3) and a complete reference implementation of the algorithms in Section 4.1, in ~10,000 lines of TypeScript and ~1,000 lines of Kotlin. `SegmentUI`, `Similarity`, and `RespPatterns` are implemented as modules and can be flexibly replaced by other implementations.

The Rx recorder traces the events by running the app under record in a VirtualXposed [57] container, which does not require any root access to the Android system. It hooks system-level APIs [2] (e.g., `Activity#dispatchTouchEvent`) to trace runtime events right before an event is consumed by the app. The Rx replayer depends only on standard Android tools UI Automator [7] to capture GUI layouts and Android debugging bridge (adb) [3] to exercise GUI events.

5 EVALUATION

Our evaluation is designed around the following two research questions:

RQ1 *To what extent does Rx push forward the state-of-the-art of cross-device record and replay?*

RQ2 *What is the time and space (logging) overhead of Rx on cross-device record and replay?*

RQ3 *What are the causes of replay failures for Rx?*

Table 3. Case study results of Rx over top commercial apps adopting non-trivial responsive patterns

ID App (Scenario)	Cross-Device Replay		
#1 GC (Simple Calc)	A ↔ B	A ↔ C	B ↔ C
#2 GC (Advanced Calc)	A ↔ B	A ↔ C	B ↔ C
#3 GC (Show History)	A ↔ B	A ↔ C	B ↔ C
#4 GC (Show Fraction)	A ↔ B	A ↔ C	B ↔ C
#5 GC (Show His. Fra.)	A ↔ B	A ↔ C	B ↔ C
#6 YT (Explore App)	A ↔ B	A ↔ C	B ↔ C
#7 YT (Enable Dark Theme)	A ↔ B	A ← C	B ← C
#8 YT (Browse Trending)	A ↔ B	A → C	B → C
#9 YT (Subscribe TBS)	A ↔ B	A ↔ C	B ↔ C
#10 MST (Goto Categories)	A ↔ B	A ↔ C	B ↔ C
#11 MST (New+Del. Task)	A ↔ B	A ↔ C	B ↔ C
#12 MST (Accom. Task)	A ↔ B	A ↔ C	B ↔ C
#13 MST (Edit Task)	A ↔ B	A ↔ C	B ↔ C
#14 MST (Del. List)	A ↔ B	A ↔ C	B ↔ C
#15 MSO (Goto Folders)	A ···· B	A ···· C	B ···· C
#16 MSO (Check Email)	A ↔ B	A ↔ C	B ↔ C
#17 MSO (Don't Disturb)	A ↔ B	A ↔ C	B ↔ C
#18 MSO (Filter Email)	A ↔ B	A ↔ C	B ↔ C
#19 MSO (Mark Email)	A ↔ B	A ↔ C	B ↔ C
#20 MSW (Open Doc)	A ↔ B	A ↔ C	B ↔ C
#21 MSW (Ins.+Del. Table)	A ↔ B	A ↔ C	B ↔ C
#22 MSW (Ren. Doc)	A ↔ B	A ↔ C	B ↔ C
#23 MSW (New+Save Doc)	A ↔ B	A ···· C	B ···· C
#24 MSW (Goto Settings)	A ↔ B	A ↔ C	B ↔ C
#25 AAR (Check ToUP)	A ↔ B	A ···· C	B ···· C
#26 AAR (Open Welcome)	A ↔ B	A ↔ C	B ↔ C
#27 FF (Nav. To Google)	A ↔ B	A ↔ C	B ↔ C
#28 FF (New Privacy Tab)	A ↔ B	A ↔ C	B ↔ C
#29 FF (Delete Browser Data)	A ···· B	A ···· C	B ···· C
#30 FF (Change Site Perm.)	A ↔ B	A ↔ C	B ↔ C
Summary	93.3%	83.3%	83.3%

Arrow direction indicates a replay success. For example, A → C denotes that a trace record on A is successfully replayed on C but not vice versa; A ↔ C denotes that the replay is successful in both directions.

5.1 Methodology

Experimental Subjects. To answer these two questions, we evaluate Rx using two sets of top commercial apps from Google Play Store [22] and conduct record and replay on three typical emulated Android devices of different screen sizes and densities:

(1) Pixel XL Phone (A): Portrait, 1440 × 2560, 560dpi,
(2) Pixel 3 XL Phone (B): Portrait, 1440 × 2960, 560dpi,
(3) Pixel C Tablet (C): Landscape, 2560 × 1800, 320dpi.

The first set of apps for evaluation (first half of Table 2) are top-downloaded apps that adopt non-trivial responsive patterns, whose replay should be sensitive to GUI restructuring and is *out of* the capability of any known available (non-search-based) record-and-replay technique. This list contains heavy apps (e.g., MS Word and YouTube) and interesting non-trivial apps (e.g., Android's official Calculator app, which adopts a non-trivial *reveal* responsive

Table 4. Experimental results of comparing Rx with RERAN and SARA

ID App (Scenario)	Same-Device Replay			Cross-Device Replay					
	Rx	RERAN	SARA	Rx			SARA		
#31 DM (Create New Canvas)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
#32 DM (Rate As Like)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
#33 ASP (Goto Craft)	A B C	A B C	A · ·	A ↔ B	A ↔ C	B ↔ C	A ← B	A ···· C	B ···· C
#34 ASP (Switch Tabs)	A B C	A B C	A · ·	A ↔ B	A ↔ C	B ↔ C	A ← B	A ← C	B ···· C
#35 ASP (Search Sea)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
#36 ASP (Create Post)	A B C	A B C	· · ·	A ↔ B	A ← C	B ← C	A ···· B	A ···· C	B ···· C
#37 ASP (Edit Post)	A B C	A B C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#38 ZDG (Goto Feat. - G.I.U)	A B C	· B C	· · ·	A ↔ B	A ← C	B ← C	A ···· B	A ···· C	B ···· C
#39 ZDG (Goto Cate. - Tech.)	A B C	A · C	· · C	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B → C
#40 ZDG (Search Video W.P.)	· · ·	A B C	· · ·	A ···· B	A ···· C	B ···· C	A ···· B	A ···· C	B ···· C
#41 CM (Explore Shop)	A B C	· B C	A B ·	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B → C
#42 CM (Change Settings)	A B C	A B C	· · C	A ↔ B	A ↔ C	B ↔ C	A ···· B	A → C	B → C
#43 CM (Make Grid)	· · ·	A B C	· · ·	A ···· B	A ···· C	B ···· C	A ···· B	A ···· C	B ···· C
#44 CM (Make F.S.)	· · ·	· · ·	· · ·	A ···· B	A ···· C	B ···· C	A ···· B	A ···· C	B ···· C
#45 CM (Make M.F.)	· · ·	· · ·	· · ·	A ···· B	A ···· C	B ···· C	A ···· B	A ···· C	B ···· C
#46 CA (Random Nav.)	A B C	A B ·	· B C	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#47 CA (Explore Home Medi.)	A B C	A · C	· · ·	A ↔ B	A ← C	B ← C	A ···· B	A ···· C	B ···· C
#48 CA (Search Music Love)	A B C	A B C	A · C	A ← B	A ↔ C	B → C	A → B	A ↔ C	B ← C
#49 CA (Change Settings)	A B C	A B C	· B C	A ← B	A ↔ C	B ↔ C	A → B	A → C	B ↔ C
#50 ADB (Explore App)	A B C	A B C	· B C	A ↔ B	A ↔ C	B ↔ C	A → B	A → C	B ↔ C
#51 ADB (Search Lovecraft)	A B C	A · C	· · ·	A ↔ B	A ← C	B ↔ C	A ···· B	A ···· C	B ···· C
#52 ADB (Filter Lovecraft)	A B C	A · C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#53 KJB (Explore App)	A B C	A · C	· · C	A ↔ B	A ← C	B ← C	A ···· B	A → C	B → C
#54 KJB (Search Love)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
#55 KJB (Edit Font Settings)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
#56 WT (Explore App)	A B ·	A B C	A B C	A ↔ B	A ← C	B ← C	A ↔ B	A ↔ C	B ↔ C
#57 WT (Choose Ori.:Sup.)	A B C	A · C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#58 WT (Sort Canvas)	A B C	A B C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#59 WT (Check App Ver.)	A B C	A · C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#60 ES (Explore App)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A → C	B → C
#61 ES (Search Lakers)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
#62 DC (Explore App)	A B C	A · C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#63 DC (Send Message)	A B C	A B C	A B ·	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ← C	B ← C
#64 DC (Check Chan. Info)	A B C	A B C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#65 DC (Set Status)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A → C	B → C
#66 UD (Explore App)	A B C	A B C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#67 UD (Chat With Diane)	· B C	A · ·	· · ·	A ↔ B	A ↔ C	B → C	A ···· B	A ···· C	B ···· C
#68 RM (Explore App)	· · ·	A B C	· · ·	A ···· B	A ···· C	B ···· C	A ···· B	A ···· C	B ···· C
#69 RM (Create Class)	· · ·	A B C	· · ·	A ···· B	A ···· C	B ···· C	A ···· B	A ···· C	B ···· C
#70 RM (Edit Class)	· · ·	A B C	· · ·	A ···· B	A ···· C	B ···· C	A ···· B	A ···· C	B ···· C
#71 SP (Explore App)	A B C	A · C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#72 SP (Browse Hip Hop)	A B C	· · C	· · ·	A ← B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#73 SP (Search RapCaviar)	A B C	A B C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#74 SP (Cre.+Del. Playlist)	A B C	A B C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#75 RD (Explore App)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
#76 RD (Join Typescript Leave)	A B C	A B C	· · ·	A ↔ B	A ↔ C	B ↔ C	A ···· B	A ···· C	B ···· C
#77 RD (Check Andr. Comm.)	A B C	A B C	A B C	A ↔ B	A ↔ C	B ↔ C	A ↔ B	A ↔ C	B ↔ C
Summary	83.7%	84.4%	35.5%	81.9%	78.7%	77.7%	33.0%	31.9%	31.9%

For same-device replay, a colored device icon (A / B / C) indicates a replay success, and a dotted gray device icon (· / · / ·) indicates a replay failure. For cross-device replay, arrow direction indicates a replay success. For example, A → C denotes that a trace record on A is successfully replayed on C but not vice versa; A ↔ C denotes that the replay is successful in both directions.

pattern). To the best of our knowledge, all known existing R&R$_c$ techniques (including those not evaluated) will fail in cross-device replaying Calculator (Figure 1).

The second set of apps for evaluation (second half of Table 2) are top-downloaded apps whose replay falls into existing work's scope. Particularly, we select the state-of-the-art open-source record and replay tools RERAN [21] (for single-device record and replay) and SARA [23] (for both single- and cross-device record and replay) as our comparison baselines[4]. To ensure a fair comparison within SARA's scope (SARA only supports the simplest *expand* responsive pattern), we adopted the following filtering process to select the second set of experimental apps: (1) Top-1 or top-2 apps of each category ranked by AppBrains [9] are selected; (2) Any app that is beyond the expand responsive pattern or incompatible with emulated environments is excluded. We downloaded these apps from ApkCombo [8], and the filtering process yielded 28 apps. To ensure a fair comparison, we also excluded 14 apps that SARA failed to parse, finally yielding the set of 14 apps for evaluation[5].

For each evaluated app, we followed existing work [23, 30] to select and create usage scenarios. Particularly, we created 3–5 scenarios for each app that represent its most common functionalities. We list them in Column 1 of Table 3 and Table 4. Readers may refer to our website for detailed information on all experimental subjects, including detailed descriptions, reproduction steps, and key events of each usage scenario.

Answering RQ1. The first part of the evaluation concerns *whether Rx is practically useful in cross-device replay.* Particularly, we conducted a case study and evaluated Rx on the 30 usage scenarios of the first 7 apps in Table 2, each consisting of six runs: $\boxed{A} \leftrightarrow \boxed{B}$, $\boxed{A} \leftrightarrow \boxed{C}$, and $\boxed{B} \leftrightarrow \boxed{C}$. Specifically, we record a usage scenario on each device (phone or tablet) and replay them on the other two. For this set of apps, we did not compare Rx with any existing technique because the replay goes *beyond* the capabilities of them. We tried to run SARA and RERAN on these cases but they failed for almost every usage scenario.

The second part of the evaluation *compares the effectiveness of Rx with existing record and replay techniques within their scopes.* This involves two sub-cases:

(1) *Same-device replay*, in which we record and replay a usage scenario on the same device ($\boxed{A}$, $\boxed{B}$, and $\boxed{C}$), over the 47 usage scenarios of the 14 subjects for comparison. All three techniques RERAN, SARA, and our Rx are evaluated.
(2) *Cross-device replay of the simplest expand responsive pattern*, which falls into the scope of SARA. In this experiment, the same 47 usage scenarios are reused, each with six replay settings: $\boxed{A} \leftrightarrow \boxed{B}$, $\boxed{A} \leftrightarrow \boxed{C}$, and $\boxed{B} \leftrightarrow \boxed{C}$. In this R&R$_c$ case, we only compare Rx with SARA because RERAN is designed for same-device replay and failed for almost every usage scenario.

Following our analysis in Section 2.3, we adopt the following oracle to determine the success of cross-device replay for both parts of the evaluation:

(1) Objectively, all key events are replayed in order;
(2) Subjectively, we confirm that the "goal" of the usage scenario is successfully achieved.

For the same-device replay of the second part, we follow existing work [12, 21, 23, 30] to determine a replay a success if (1) all events are replayed in order and (2) all visual state transitions meet a human developer's expectation.

Answering RQ2. We collect record/replay time statistics and compare them with the original execution (without our instrumentation) for each Rx's successful cross-device replay to evaluate the runtime overhead of Rx. We also collect the disk usage of all logs (without compression) generated by Rx to measure the space overhead.

Answering RQ3. We manually inspect failed cases and categorize their root causes. The detailed analyses can be found in Section 5.4.

Experimental Environments. Overall, the experiments consist of 1,167 different replay settings (technique × usage scenario × device). We observed negligible flakiness in the replayed usage scenarios, and we consider a replay success if *two* consecutive replays both satisfied the replay oracle. All experiments were conducted on a quad-core Intel i7-7700 desktop with 32GiB RAM running Ubuntu 20.04.1 LTS, with Android API 27. Rx and the two baselines are evaluated under the same emulated hardware/software configuration.

5.2 Evaluation Results for RQ1: Effectiveness

Case Study. Table 3 displays the case study results. This experiment involves 30 usage scenarios with each replayed six runs: $\boxed{A} \leftrightarrow \boxed{B}$, $\boxed{A} \leftrightarrow \boxed{C}$, and $\boxed{B} \leftrightarrow \boxed{C}$, yielding 180 replay runs in total. Overall, Rx succeeded in 86.7% (156/180), in which 80.0% (24/30) scenarios succeeded in all six runs.

We would like to emphasize that these usage scenarios are out of any known (non-search-based) technique's capabilities. For example, SARA [23] attributes its replay failure of Calculator (resembles $\boxed{A} \rightarrow \boxed{B}$) to "breaking UI changes". However, this is a reveal responsive pattern recommended by the Android's official guideline. On the other hand, Rx succeeded in all five usage scenarios of the Calculator app. Readers may refer to our website for further information, including a video of a $\boxed{C} \rightarrow \boxed{A}$ replay for Calculator, where cross-device replaying the event trace (consisting of 11 clicks and 1 swipe) on $\boxed{A}$ yields a different trace of 19 clicks and 1 swipe to make the replay successful.

Same-Device Record and Replay. The "Same-Device Replay" column in Table 4 displays the evaluation results. We compare Rx, RERAN, and SARA using the scenarios #31–#77 in Table 4. We record and replay each scenario on the same device ($\boxed{A}$, $\boxed{B}$, $\boxed{C}$), yielding 141 replay runs for each evaluated technique.

Rx succeeded in 83.7% (118/141) replay runs. The numbers are 84.4% (119/141) for RERAN and 35.5% (50/141) for SARA. Overall, Rx has a competitive successful replay rate with the pixel-precise, state-of-the-art same-device replay technique RERAN, even though the Rx replayer has a much more complex workflow.

[4] For closed-source tools, we contacted the authors of RANDR [48] and V2S [12] for tool binaries but received no response. We also tried to compare with appetizer [10], which was state-of-the-practice. Unfortunately, we have to exclude it because it was extremely unstable and frequently lost events.

[5] However, we also evaluated the excluded 14 apps (only on Rx and RERAN), and the experimental results are available on our website. Overall, Rx received a 78.3% and 82.5% successful rate in cross- and single-device record and replay, respectively, while RERAN obtained a 74.4% single-device successful rate.

On the other hand, Rx is the most stable tool in the evaluation that produces consistent results over devices. For Rx, 80.9% (38/47) of the usage scenarios were successfully replayed on all three devices. The numbers are 66.0% (31/47) for RERAN and 23.4% (11/47) for SARA. Rx has better stability than RERAN because replaying raw events in RERAN suffers from minor non-deterministic device behaviors (e.g., scrolling has small non-determinism across replay runs). In contrast, Rx's semantic-aware matching mechanism can tolerate such behaviors. Furthermore, the analyses in Section 5.4 show that the 9 failure cases for app CollageMaker are due to missing accessibility information.

It is a surprise that SARA failed in so many cases. SARA frequently froze (or even crashed) at record time due to flushing large amounts of logs[6]. The offline self-replay of SARA also contributed to many replay failures. The self-replay of SARA is expected to be a faithful same-device replay used to translate raw information (e.g., coordinate) to views. However, self-replaying a scrolling event may unfaithfully yield different scroll distances due to minor non-determinism of UI, and SARA's replay thereby fails if self-replay diverges. Rx does not suffer from this because Rx grabs the view information when recoding and does not need a self-replay phase to translate.

Cross-Device Record and Replay. The "Cross-Device Replay" column in Table 4 displays the evaluation results. This experiment is based on the previous 47 usage scenarios (#31–#77) created for same-device replay. We excluded RERAN in this experiment because replaying raw input events on another device of a different screen makes nonsense. Each scenario consists of six runs, yielding 282 replay runs in total.

Rx succeeded in 79.4% (224/282) cases, in which 63.8% (30/47) scenarios succeeded in all the six replay runs. Despite that all these usage scenarios have been carefully chosen to fit into SARA's scope, the numbers for SARA are 32.3% (91/282) and 19.1% (9/47), resulting from similar failure causes to the same-device replay.

5.3 Evaluation Results for RQ2: Overhead

The performance evaluation results are summarized as follows (and detailed results are available in the supplementary materials):

(1) The record-time logging costs 372ms and 23KiB of log (without compression) per event on average. This includes the time (~300ms) to capture GUIs and space (~23KiB) saving GUI's layout dump. Such a cost is required by any GUI layout based replay technique [5, 20, 23, 48, 58].
(2) The replay-time UI segmentation, responsive pattern matching, and responsive action execution cost 411ms on average, where applying a responsive action occupies ~350ms.
(3) For events whose receiver does not exist at the replay time, averagely 1.6 triggering events are additionally performed to expose the key event receiver on the replay device.

Overall, the runtime and space overhead is considered affordable for a human operator/replayer, indicating that our Rx implementation enabled practical R&R$_c$ uses, e.g., cross-device unit testing for an app developer.

Compared with existing event-based replay tools which are not sensitive to GUI restructuring and assume any event e's receiver should exist at the replay time [21, 23, 48], Rx pays only 7.31% time overhead and 0.2 additional (triggering) events to gain the *cross-device* replay capability towards GUI restructuring. This result is consistent with the least surprise principle in GUI design that a human should be able to find the key event with least GUI perturbation. Moreover, capturing GUI layouts and executing events occupy ~80% of the runtime overhead. This conforms to Wang et al.'s study [58] and thereby the performance of Rx can be further improved by replacing UI Automator with Toller.

The average log size of Rx is 23KiB (3.3KiB) per event and 198KiB (29KiB) per usage scenario without (with) compression [71]. Such a low[7] space usage is within our expectation because Rx's logging is intentionally designed to use as less disk space as possible while record the GUI dump of every event. Specifically, Rx's log maintains a pool to deduplicate strings and views. Each view (and view's property) in the GUI layout is represented by its index in the pool, and the GUI layout tree is then serialized as a flattened index sequence by the pre-order of residing views.

5.4 Evaluation Results for RQ3: Failure Cases

Though largely outperforming existing work, Rx still failed at 105 out of 603 replays (17.4%), including both single- and cross-device replay cases affecting 23 usage scenarios of 15 apps in Tables 3 and 4. We analyze them in detail to shed light on future improvements. The visual convention is that an italic text denotes the cause of a replay failure, and underlined numbers denote the case counts.

Framework Limitations (2/105, 1.9%). All event-based replay algorithms (including Rx) assume that both record and replay are conducted in a deterministic environment. However, a perfectly deterministic environment simply does not exist. Even though the experiments demonstrated that Rx can better handle minor UI non-determinism than a pixel-precise replay mechanism (Rx produced consistent and stable results over devices in the same-device replay experiments), apps may contain *dynamic contents* (2) that change over time (e.g., real-time news feeds) and result in replay failures. Similar to prior work [5, 6, 10, 20, 23, 25, 55], Rx is limited in replaying pixel-precise complex gestures (e.g., pinch) and network traffic. Considering that both deterministic replay (usually for failure reproduction) and pixel-precise replay are out of the scope of event-based replay, we believe that the Rx framework has well-supported cross-device record and replay.

Implementation Limitations (60/105, 57.1%). Rx framework relies on the interfaces `SegmentUI`, `Similarity`, and `RespPatterns` that are implementation-specific for performing human-related UI understanding. Our current implementation is limited and results in some replay failures.

The most frequent failure cause that affects our `SegmentUI` implementation is *overlapping views* (30), where a horizontal/vertical

[6] Our evaluation was conducted on an Android Emulated Device with hardware virtualization over a mainstream desktop Intel i7 processor, which is significantly faster than a mobile processor. Therefore, the performance issue of SARA should be considered an implementation limitation.

[7] Should note that our evaluation includes apps that are deemed to have complicated GUIs, e.g., MS Word, ESPN, and Webtoon.

split cannot yield a correct partition. In these cases, a correct segmentation must "cut through" at least one view, which is strictly prohibited in Algorithm 2.

Our RespPatterns are also limited in identifying particular patterns. For example, the $P_{\text{ex-sl}}$ pattern tries a fixed percentage of scrolling offset for a scrollable list. However, sometimes *unstable scroll* (4) events may cause the replayer to over-scroll the list and miss the target event. Other related failure causes include failing to model a pattern (8) and failing to recognize a known pattern (6). Considering that our tool is designed to be extensible, these limitations can be mitigated by either adding new patterns or rewriting existing patterns for adapting to specific apps.

Finally, our tool implementation has its limitations (12). For example, Rx does not support self-rendered apps (e.g., WebView and game-engine empowered apps) by far because UI Automator fails to dump them.

App Bugs (43/105, 41.0%). To our surprise, the basic usage scenarios for experimental evaluation even revealed *functional bugs* (4) in these top commercial apps. A crash bug is from AdobeReader (usage scenario #25, specific to C, causing 4 replay failures), which non-deterministically hits a null-pointer deference in the app's native library `libADCComponents.so`. The triggering is non-deterministic, potentially due to concurrency issues. Another non-crash bug is from KingJamesBible (usage scenario #55, all devices) where the view displaying the font of the current text failed to refresh over font changes. Considering that the latter bug did not affect replay, we still consider it a replay success in Table 4.

The final major cause of replay failures is app's *accessibility bugs* (39) because our Rx Similarity and RespPatterns are based on precise accessibility information, particularly view's textual descriptions that enable visually impaired people to use the phone with text-to-speech technologies. These accessibility bugs can be further classified into *incomplete* (33) or *incorrect* (6) accessibility information. Rx cannot conduct correct UI segmentation and responsive pattern recognition on empty accessibility information (like CollageMaker). Rx can be erroneously trapped into an unexpected replay state on incorrect accessibility information (like MS Outlook and Remind). For a similar reason, people with vision impairment will suffer from using these apps. In this sense, replay success is expected if Similarity and RespPatterns do not depend on accessibility information.

We reported the accessibility bug in MS Outlook (with usage scenario #15, in which "close"/"open" is labeled in the opposite), the only evaluated app that provides an explicit in-app bug report mechanism[8]. We received positive confirmation from the developers, and they claimed a bug fix in version 4.21.3.

5.5 Discussions

RQ1&2: Effectiveness and Overhead of the Rx Framework. Overall, Rx succeeded in 498 of 603 replays (82.6%) with affordable overhead. The evaluation results indicate that Rx well handles cross-device replay cases over apps that have GUI restructuring. For usage scenarios that are within existing work's scope, Rx gains a competitive or better capability than state-of-the-art techniques. Furthermore, Rx's results are considerably more consistent and stable across devices.

RQ3: A Call for Practical R&R$_\text{c}$. The analyses of failure cases first revealed that only very few replay failures (less than 1% in all replays) are out of the capability of the Rx replay framework (Algorithm 1). Perhaps this is our most significant implication: *event-based replay, even for the challenging cross-device case, can be done greedily online.*

Considering the other failure cases, this paper could be regarded as *a call for future research along R&R$_\text{c}$*. It is expected that a heuristic algorithm (like the one in Section 4.1) cannot handle all real-world cases: overlapping views cause SegmentUI to perform incorrect segmentation, and the quality of accessibility information significantly impacts Similarity and RespPatterns. These limitations can be potentially resolved by data-driven approaches, e.g., by learning distributed representations of views. We are optimistic that learning-based approaches, e.g., V2S [12], will facilitate better implementations of SegmentUI, Similarity, and RespPatterns.

Threats to Validity. The first threat to validity is that only the most popular apps are selected in the evaluation. Most of them are commercial apps developed by professional teams. Therefore, the evaluation results may not generalize to other apps, e.g., less well-maintained open-source apps. We intentionally chose these most popular apps because they usually incorporate complex patterns in adapting to different screen sizes. The major conclusion concerning the effectiveness of Rx replay framework should remain positive on simpler apps with less complex responsive patterns[9].

Another threat is that all usage scenarios are created by the authors, which may be subject to bias. To best alleviate this issue, we tried our best to select the most typical usage scenarios by following existing work [23, 30]. We also provide detailed descriptions of these usage scenarios on our website to enable reproducible research in the future.

The final threat is that the replay success is determined by a human, even though such a human in the loop seems unavoidable. The analyses in Section 2 show that key events are a good basis for determining replay success. We strictly followed this guidance as the replay oracle to avoid bias from humans.

6 RELATED WORK

Record and replay is a fundamental enabling technology for a broad spectrum of testing and debugging practices and has been extensively studied in various contexts.

Record and Replay for Android Apps. RERAN [21] exploits Linux's input subsystem (`/dev/input/event*`) for logging and replaying pixel-precise hardware-level input events, in which views are located by absolute coordinates. Pixel-preciseness is a desirable property for deterministic apps. However, as shown in our experiments, even minor non-determinism can result in replay failure. Other pixel-based approaches include VALERA [26] (a customized system) and Mobiplay [44] (record and replay on a remote-desktop).

Such a pixel-precise treatment can easily fail to replay a trace on a device of different screen size or orientation. To enable cross-device

[8] Accessibility bug is a well-known source of Android app's bugs, and thus we did not report the other bugs. Regarding more apps in the Play Store, approximately 45% of GUIs have at least one ImageButton that misses accessibility textual description [16].

[9] However, Rx does not well support apps that refuse a correct application of officially recommended responsive patterns.

replay to some extent, one may proportionally scale the coordinates [6, 10, 12, 25, 55, 65] or localize a logged event e's receiver object (at replay time) by its attributes [5, 20, 23, 38, 48, 53, 55, 56, 70], e.g., using textural information of views [5, 20, 23, 35, 48]. However, all these techniques made a fundamental assumption that each logged event e's receiver object exists on the replay-time GUI. Unfortunately, modern apps often adopt Android's responsive design that could extensively restructure the UI layout across devices (usage scenarios #48-77). This paper is the first to consider these cases as in-scope.

Finally, as discussed in Section 2, one may incorporate an exhaustive search for an oracle-satisfying event sequence [11, 31, 40, 42, 43, 59, 63, 69]. A sufficiently efficient search will ultimately solve R&R_c and many other problems (e.g., test input generation) and is still an open problem. On the other hand, this paper as an exploratory demonstrates that search may not be required for a practical R&R_c.

Record and Replay for Other Event-Based Systems. There is record and replay work for other event-based systems, e.g., Web applications. Web applications have a simpler single-threaded event model, whose execution is easier to be deterministically traced [1, 37, 47, 52, 66]. Upon that, a developer-friendly interactive debugger [14] or further sophisticated dynamic analyses [51] can be implemented. Generally, Web applications are simpler in the execution model and responsive patterns, and have a particular focus on the network side [41]. Thus, the record and replay of Web application is a considerably different scope.

Deterministic Replay for Other Systems. A general program's execution can also be made fully (or partially) deterministic via runtime tracing. Such deterministic replay techniques may involve input tracing [46], instruction tracing [19], or memory tracing [27]. These (deterministic) approaches could incur significantly higher overhead and are more intrusive than a lightweight event-based record and replay (this paper's scope). We do not discuss them further because they target a different replay purpose.

GUI Understanding and Analyses. Finally, the implementations of UI segmentation and responsive patterns are related to GUI understanding, e.g., learning probabilistic distribution of GUI layouts for detecting and repairing GUI bad-smells and flaws. UIS-Hunter [64] and Seenormly [67] exploit computer-vision techniques to detect violations with respect to Material Design's don't-do-that guidelines. LabelDroid [16] and COALA [36] repair missing textual descriptions for `ImageView`-like views using data-driven approaches. Gvt [39] and OwlEye [32] detect gaps between GUI designs and their implementations. We are optimistic that these techniques, orthogonal to this paper's scope, can facilitate further development of cross-device record and replay technologies.

7 CONCLUSION

Record and replay is a foundational technology for a broad spectrum of Android app testing and debugging practices. This paper made the cross-device record and replay practical for industrial-scale apps with respect to responsive GUI restructurings by leveraging the principle of least surprise in GUI design, i.e., spatial locality and responsive patterns, with promising experimental evaluation results. We hope this work, which pushes forward the state-of-the-art of cross-device record and replay for Android apps, will serve as a call for future research along this line.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the Natural Science Foundation of China under Grant Nos. 61932021 and 62025202, and the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001. The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn) and Chang Xu (changxu@nju.edu.cn) are the corresponding authors.

REFERENCES

[1] Silviu Andrica and George Candea. 2011. WaRR: A Tool for High-Fidelity Web Application Record and Replay. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems Networks (DSN '11)*. 403–410. https://doi.org/10.1109/DSN.2011.5958253

[2] Android. 2021. *Android API Reference*. https://developer.android.com/reference/

[3] Android. 2021. *Android Debug Bridge (adb)*. https://developer.android.com/studio/command-line/adb

[4] Android. 2021. *Android Fragments*. https://developer.android.com/guide/fragments

[5] Android. 2021. *Espresso*. https://developer.android.com/training/testing/espresso

[6] Android. 2021. *monkeyrunner*. https://developer.android.com/studio/test/monkeyrunner

[7] Android. 2021. *UI Automator*. https://developer.android.com/training/testing/ui-automator

[8] ApkCombo. 2021. *ApkCombo*. https://apkcombo.com

[9] AppBrain. 2021. *Google Play Ranking: The Top Free Overall*. https://www.appbrain.com/stats/google-play-rankings

[10] Appetizer. 2021. *appetizerio/replaykit*. https://github.com/appetizerio/replaykit

[11] Farnaz Behrang and Alessandro Orso. 2019. Test Migration between Mobile Apps with Similar Functionality. In *Proceedings of the 2019 IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. 54–65. https://doi.org/10.1109/ASE.2019.00016

[12] 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 2020 ACM/IEEE International Conference on Software Engineering (ICSE '20)*. 309–321. https://doi.org/10.1145/3377811.3380328

[13] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1987. Occam's razor. *Information processing letters* 24, 6 (1987), 377–380.

[14] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 2013 Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. 473–484. https://doi.org/10.1145/2501988.2502050

[15] Deng Cai, Shipeng Yu, JiRong Wen, and WeiYing Ma. 2003. VIPS: a Vision-based Page Segmentation Algorithm. (2003).

[16] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *Proceedings of the 2020 ACM/IEEE International Conference on Software Engineering (ICSE '20)*. 322–334. https://doi.org/10.1145/3377811.3380327

[17] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. 2018. DetReduce: Minimizing Android GUI Test Suites for Regression Testing. In *Proceedings of the 2018 International Conference on Software Engineering (ICSE '18)*. 445–455. https://doi.org/10.1145/3180155.3180173

[18] Material Design. 2021. *Responsive Patterns*. https://material.io/archive/guidelines/layout/responsive-ui.html#responsive-ui-patterns

[19] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI '02)*. 211–224. https://doi.org/10.1145/844128.844148

[20] Mattia Fazzini, Eduardo Noronha De A. Freitas, Shauvik Roy Choudhary, and Alessandro Orso. 2017. Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests. In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST '17)*.

149–160. https://doi.org/10.1109/ICST.2017.21

[21] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 72–81. https://doi.org/10.1109/ICSE.2013.6606553

[22] Google. 2021. *Google Play Store*. https://play.google.com/store

[23] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. Sara: Self-Replay Augmented Record and Replay for Android in Industrial Cases. In *Proceedings of the 2019 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. 90–100. https://doi.org/10.1145/3293882.3330557

[24] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 2008 USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. 193–208. https://doi.org/10.5555/1855741.1855755

[25] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay J. Reddi. 2015. Mosaic: Cross-Platform User-Interaction Record and Replay for The Fragmented Android Ecosystem. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '15)*. 215–224. https://doi.org/10.1109/ISPASS.2015.7095807

[26] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. 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 (OOPSLA '15)*. 349–366. https://doi.org/10.1145/2814270.2814320

[27] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight Deterministic Multi-Processor Replay of Concurrent Java Programs. In *Proceedings of the 2010 ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. 385–386. https://doi.org/10.1145/1882291.1882361

[28] Geoffrey James. 1987. Law of Least Astonishment. *The Tao of Programming* (1987).

[29] Karen Sparck Jones. 1972. A Statistical Interpretation of Term Specificity and Its Application in Retrieval. *Journal of documentation* (1972).

[30] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and Replay for Android: Are We There yet in Industrial Cases?. In *Proceedings of the 2017 Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. 854–859. https://doi.org/10.1145/3106237.3117769

[31] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *Proceedings of the 2019 IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. 42–53. https://doi.org/10.1109/ASE.2019.00015

[32] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *Proceedings of the 2020 IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. 398–409. https://doi.org/10.1145/3324884.3416547

[33] Hans Peter Luhn. 1957. A Statistical Approach to Mechanized Encoding and Searching of Literary Information. *IBM Journal of research and development* 1, 4 (1957), 309–317.

[34] Baoying Ma, Li Wan, Nianmin Yao, Shuping Fan, and Yan Zhang. 2020. Evolutionary Selection for Regression Test Cases Based on Diversity. *Frontiers of Computer Science* 15, 2 (2020), 2095–2236.

[35] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic Matching of GUI Events for Test Reuse: Are We There Yet?. In *Proceedings of the 2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*. 177–190. https://doi.org/10.1145/3460319.3464827

[36] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-Driven Accessibility Repair Revisited: On the Effectiveness of Generating Labels for Icons in Android Apps. In *Proceedings of the 2021 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. 107–118. https://doi.org/10.1145/3468264.3468604

[37] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 2010 USENIX Conference on Networked Systems Design and Implementation (NSDI '10)*. 11. https://doi.org/10.5555/1855711.1855722

[38] Kevin Moran, Richard Bonett, Carlos Bernal-Cárdenas, Brendan Otten, Daniel Park, and Denys Poshyvanyk. 2017. On-Device Bug Reporting for Android Applications. In *Proceedings of the 2017 International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. 215–216. https://doi.org/10.1109/MOBILESoft.2017.36

[39] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated Reporting of GUI Design Violations for Mobile Apps. In *Proceedings of the 2018 International Conference on Software Engineering (ICSE '18)*. 165–175. https://doi.org/10.1145/3180155.3180246

[40] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST '16)*. 33–44. https://doi.org/10.1109/ICST.2016.34

[41] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. 417–429. https://doi.org/10.5555/2813767.2813798

[42] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2020. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3007664

[43] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: Migrating GUI Test Cases from IOS to Android. In *Proceedings of the 2019 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. 284–295. https://doi.org/10.1145/3293882.3330575

[44] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. In *Proceedings of the 2016 International Conference on Software Engineering (ICSE '16)*. 571–582. https://doi.org/10.1145/2884781.2884854

[45] Eric Steven Raymond. 2003. Applying the Rule of Least Surprise. *The Art of Unix Programming* (2003).

[46] rr. 2021. *rr*. https://rr-project.org

[47] rrweb. 2021. *rrweb: Record and Replay the Web*. https://www.rrweb.io

[48] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse K. Coskun, and Manuel Egele. 2019. RandR: Record and Replay for Android Applications via Targeted Runtime Instrumentation. In *Proceedings of the 2019 IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. 128–138. https://doi.org/10.1109/ASE.2019.00022

[49] Yasushi Saito. 2005. Jockey: A User-Space Library for Record-Replay Debugging. In *Proceedings of the 2005 International Symposium on Automated Analysis-Driven Debugging (AADEBUG '05)*. 69–76. https://doi.org/10.1145/1085130.1085139

[50] Peter Seebach. 2001. The Cranky User: The Principle of Least Astonishment. *IBM DeveloperWorks* (2001).

[51] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. 488–498. https://doi.org/10.1145/2491411.2491447

[52] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. 2005. Automated Replay and Failure Detection for Web Applications. In *Proceedings of the 2005 IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. 253–262. https://doi.org/10.1145/1101908.1101947

[53] CulebraTester Team. 2021. *CulebraTester*. http://culebra.dtmilano.com/

[54] GDB Team. 2021. *GDB: The GNU Project Debugger*. https://www.gnu.org/software/gdb/

[55] Ranorex Team. 2021. *Ranorex*. https://www.ranorex.com/

[56] Robotium Team. 2021. *Robotium*. https://github.com/RobotiumTech/robotium

[57] tiann. 2021. *VirtualXposed*. https://github.com/android-hacker/VirtualXposed

[58] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools. In *Proceedings of the 2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*. 165–176. https://doi.org/10.1145/3460319.3464828

[59] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Generating Reproducible and Replayable Bug Reports from Android Application Crashes. In *Proceedings of the 2015 IEEE International Conference on Program Comprehension (ICPC '15)*. 48–59. https://doi.org/10.1109/ICPC.2015.14

[60] Wikipedia. 2021. *Tf–Idf*. https://en.wikipedia.org/wiki/Tf%E2%80%93idf

[61] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. 2011. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC '11)*. 29. https://doi.org/10.5555/2002181.2002210

[62] Chang Xu, Yi Qin, Ping Yu, Chun Cao, and Jian Lu. 2020. Theories and Techniques for Growing Software: Paradigm and Beyond. *SCIENTIA SINICA Informationis* 50, 11 (2020), 1595–1611.

[63] Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Tian Zhang, Yuetang Deng, and Xuandong Li. 2021. GUIDER: GUI Structure and Vision Co-Guided Test Script Repair for Android Apps. In *Proceedings of the 2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*. 191–203. https://doi.org/10.1145/3460319.3464830

[64] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. Don't Do That! Hunting Down Visual Design Smells in Complex UIs Against Design Guidelines. In *Proceedings of the 2021 IEEE/ACM International Conference on Software Engineering (ICSE '21)*. 761–772. https://doi.org/10.1109/ICSE43902.2021.00075

[65] 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 *Proceedings of the 2019 IEEE/ACM International Conference on Automated Software Engineering (ICSE '19)*. 1066–1069. https://doi.org/10.1109/ASE.2019.00103

[66] Lu Zhang and Chao Wang. 2017. RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay. In *Proceedings of the 2017 International*

Conference on Software Engineering (ICSE '17). 278–288. https://doi.org/10.1109/ICSE.2017.33

[67] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-Based Linting of GUI Animation Effects against Design-Don't Guidelines. In *Proceedings of the 2020 ACM/IEEE International Conference on Software Engineering (ICSE '20)*. 1286–1297. https://doi.org/10.1145/3377811.3380411

[68] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: A Framework for Evaluating UI Test Reuse. In *Proceedings of the 2020 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. 1190–1201. https://doi.org/10.1145/3368089.3409708

[69] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *Proceedings of the 2019 International Conference on Software Engineering (ICSE '19)*. 128–139. https://doi.org/10.1109/ICSE.2019.00030

[70] Jiahuan Zheng, Liwei Shen, Xin Peng, Hongchi Zeng, and Wenyun Zhao. 2020. MashReDroid: Enabling End-User Creation of Android Mashups Based on Record and Replay. *Science China Information Sciences* 63, 10 (2020), 1869–1919.

[71] zlib. 2021. *zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library*. https://www.zlib.net