EP-Detector: Automatic Detection of Error-prone Operation Anomalies in Android Applications

Chenkai Guo*†‖, Qianlu Wang*, Naipeng Dong‡, Lingling Fan*, Tianhong Wang§,
Weijie Zhang§, Enbao Chen*, Zheli Liu*, and Lu Yu¶‖
*College of Cyber Science, Nankai University, China
†Haihe Lab of ITAI, China
‡The University of Queensland, Australia
§College of Computer Science, Nankai University, China
¶National University of Defense Technology, China
‖Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, China
{guochenkai,wql,linglingfan,tianhongwang,2210977,liuzheli}@nankai.edu.cn, n.dong@uq.edu.au,
a505025234@gmail.com, yulu@nudt.edu.cn

***Abstract*—Android applications are pervasively adopted and heavily relied on in our daily life, leading to the growing demand for enhanced user experiences, such as ease for operation and robustness. Nevertheless, developers continue to prioritize traditional functionality and performance, overlooking the pivotal role of user experience in real-world scenarios. For example, poorly designed page elements can lead to user confusion, resulting in unexpected outcomes, termed as the *error-prone operation anomalies (EPAs)*. In this work, we undertake the first effort to uncover the underlying essence of the EPA problem. To achieve this objective, we investigated the root causes of EPAs from three dimensions, i.e., subject, object and environment. These causes were identified by multi-stage attribute capturing and precise similarity computation. In this process, the causes are categorized into fine-grained classes, namely confusing behaviours, unsuitable layout, and resource overload. Building upon these insights, we propose a dynamic GUI-based testing tool *EP-Detector* to facilitate detecting the EPAs in real-world apps. The EP-Detector is equipped with widget-exploration based target navigation and automatic test oracle, enabling it to detect error-prone page elements and simulate events with both comprehensiveness and precision. To systematically study the prevalence and severity of real-world EPAs, we conducted experiments on 53 popular Android apps with EP-Detector. The confirmed results not only validate the high precision and completeness of EP-Detector but also highlight that EPAs are prevalent in current apps, with at least one EPA existing in every two page widgets on average, and 28.3% of them may lead to security and functionality issues or risks. The EP-Detector is available at https://github.com/WordDealer/EP-Detector.**

***Index Terms*—Android application, User operation, Anomaly detection, Error-prone operation, Automated test**

I. Introduction

Android applications (apps for short) are software running on mobile systems, which are normally equipped with numerous Graphical Use Interfaces (GUIs) for miscellaneous user interactions. *User behaviours* are the typical input to Android apps, driving the execution of the apps' underlying functionality. The diversity of user behaviours and their temporal nature raise the complexity of an app's execution routines, thereby posing a challenge in GUI-based app testing. Since the pioneering work by Hu et al. [1] in 2011, the testing of GUI-based mobile apps has gradually emerged as a focal area of interest in both industry [2]–[4] and academia [5]–[7]. Traditional GUI-based testing generates test cases to approximate critical user interactive behaviours or guide the execution paths, which facilitates uncovering fatal crashes [8]–[10], framework weaknesses [11], [12], non-crashing functional bugs [13]–[15] or security vulnerabilities [16], [17]. In existing GUI-based testing works, a latent but strong assumption is that *all test cases are correctly and accurately executed by users.*

However, achieving this assumption can be challenging in the practical use of Android apps. On one hand, as user requirements for rich functionalities increase, the user events specified by Android tend to be more miscellaneous and complex. For instance, Android introduced an optional gesture system, including "swipe up", "swipe in a hook move" etc. in version 10.0 [18], which allows replacing all bottom buttons with a single bar. However, users often find these swiping operations confusing, leading to imprecise and incorrect operations. On the other hand, Android apps typically operate under stringent device limitations and within complex scenarios compared to traditional PC devices, which further amplifies the challenge of performing correct operations. For instance, the screen size of mobile devices are normally limited. Users are required to accomplish diverse operations with a bigger "cursor" (fingers) on a relatively smaller range. In addition, the operations of mobile apps can hardly be stable, due to the heterogeneous use scenarios, e.g., users usually operate the mobile device when standing, walking or even lying down. Moreover, the system resources of Android, e.g., CPU, RAM and network, are constrained, and such resource limitation tends to raise system freezes and crashes, which further increases the risk of misoperations.

The misoperations are often triggered by unreasonable design of page elements. A sound page design should be

Corresponding author: Lingling Fan.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00029

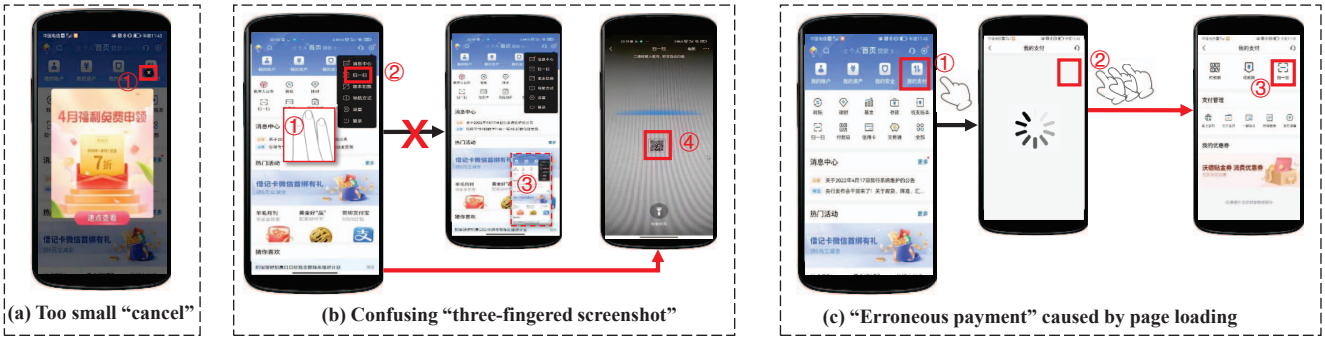


Fig. 1. Motivating Examples.

robust to the aforementioned sensitivities. That is, the user will not experience misoperations resulting from 1) inadequate interactions with widgets, 2) layout issues of widgets, or 3) resource environment constraints. In this work, we refer to the user's misoperation caused by error-prone page elements as the Error-prone Operation Anomaly (EPA). The ubiquity of EPAs (which will be verified in our later experiments) produce negative impacts on apps such as degraded user experience, execution crashes and security risks for users without professional exercise. However, to the best of our knowledge, there is still lack of systematical study and precise detection for the EPAs. The challenges of EPA detection are two-fold. First, the consequences of EPA have diverse manifestations, making it difficult to capture the underlying root-causes. Second, given the probabilistic occurrence of EPAs, it is challenging to trigger and identify various types of such unconventional misoperations in a consistent manner. In this paper, we address the challenges by proposing a novel dynamic testing technique *EP-Detector*.

To identify the root-causes of EPAs, EP-Detector analyzes each target widget that users interact with, based on three factors: *subject*, *object* and *environment*.

- The *subject* refers to user behaviours interacting with the target widget. If similar user operations yield different outcomes, termed *confusing behaviours*, and if the widget supports them, users are likely to be misled.
- The *object* refers to *design* of widgets capable of triggering error-prone operations. One prominent issue is the *unsuitable layout* in the widget, for example the elements' size and spacing, which often results in misoperations.
- The *environment* refers to the resources that facilitate the execution of the target widget such as the CPU, RAM and network. Drastic environmental changes, such as resource overload, can lead to misoperations, particularly if the widgets are sensitive to the changes.

EP-Detector then develops GUI-based testing from the above insights. To address the challenge for triggering and identifying EPAs, EP-Detector employs the following techniques: ❶ Multiple execution strategies and test oracle methods are designed to capture the heterogeneous dynamic nature of the testing. ❷ A two-stage target navigation with fine-grained trace guidance and multi-level target identification is designed and implemented to ensure a smooth and accurate detection. ❸ Optimization strategies such as page-level abstraction and operation grouping are devised to enhance detection efficiency. The performance of EP-Detector is evaluated on 53 well-known real-world apps covering diverse app categories containing 9073 widgets across 525 pages. The EP-Detector detects 5136 confirmed EPAs, with an average precision of 88.28%. The results illustrate that EPAs are prevalent in current apps, with at least one EPA existing in every two page widgets on average, and 28.3% of them may lead to security or functionality risks.

In summary, the contributions of this paper are as follows:

- We are the first to conduct a systematic study of the error-prone operation anomalies (EPAs) in Android apps, including their prevalence, root-causes and impacts.
- We propose a novel testing technique *EP-Detector* for the dynamic detection of EPAs, covering behaviour-, layout- and environment-based error-prone page elements.
- Experiments on 53 real-world apps validate the effectiveness of the EP-Detector. The experimental results reveal intriguing findings, with careful examination of typical EPA cases showcasing the impacts of the detected EPAs.

II. Motivating Examples

This section motivates our research by presenting three types of error-prone user operations within the same Android app. Illustrated in Figure 1 is the GUI of a well-known Chinese banking app, exemplifying these irreversible error operations:

- EPA 1: When a user opens the app, she will be directed to the home page. As shown in Figure 1 (a), an advertisement dialog pops up on top of the actual home page. If the user wants to close the advertisement and visit the home page, she may find that the closing button is small and somewhat hidden (as shown in (a)-①). This can result in accidentally pressing the wrong area, which redirects the user to the advertisement page instead.
- EPA 2: When the user visits the main operation page and attempts to use the three-finger swipe for taking a screenshot, provided by the Huawei Harmony to save the key page information (as shown in (b)-①), the system

may mistakenly interpret it as a click due to the excessive force of the first finger. This inadvertent click could trigger the bank scan function ((b)-②), leading the app to the scan page. If, coincidentally, a payment QR code appears in front of the camera ((b)-④), the app will be redirected to the payment transfer page.

- EPA 3: The user intends to visit the "My Payment" page (as shown in (c)-①) to set the payment limit. However, if the page loading is stuck due to for example the network delay, the user may assume it's her operation fault and repeatedly click on the screen ((c)-②). When the "My Payment" page eventually loads, these additional clicks may inadvertently trigger actions if there are buttons placed in the same position on the "My Payment" page e.g., the payment scan button (as shown in (c)-③). As a result, these unintended clicks lead the app to the payment scan and subsequently to the payment transfer page if a payment QR code is detected in front of the camera.

The above examples demonstrate that app design and implementation may lead to user's error-prone operations, which result in irreversible and unexpected consequences that significantly reduce user experience and increase security risks.

III. Problem Definition

The entire process of EPA detection can be formally represented as a 3-tuple $\mathcal{A} = \langle Q, \Sigma, \delta \rangle$, where Q is the set of execution states; Σ represents the set of events triggering state changes; and $\delta : Q \times \Sigma \rightarrow Q$ refers to the state transitions. The Σ contains three error-prone event groups E_b, E_a, E_e $(E_b \subseteq \Sigma, E_a \subseteq \Sigma, E_e \subseteq \Sigma)$ representing the EPA events caused by *behaviour*, *layout* and *environment*, respectively. Let $\mathcal{E} = E_s|E_b|E_e$ be an EPA event group; e_1 and e_2 are error-prone events within the same group $(e_1 \in \mathcal{E} \wedge e_2 \in \mathcal{E})$; $s \in Q$ is an execution state; $\delta(s, e_1) \rightarrow s_{t1}$ and $\delta(s, e_2) \rightarrow s_{t2}$ are two state transitions triggered by e_1 and e_2 respectively, the *EPA* problem can be defined as:

$$Diff\,(s_{t1}, s_{t2}) > \Gamma, \tag{1}$$

where the $Diff$ is a differential function (defined later) for computing the difference between s_{t1} and s_{t2}; Γ is a given threshold. The intuition is that if there exists e_1 and e_2 leading to states that cannot be distinguished, then EPA occurs.

We require that: ❶ An execution state $s \in Q$ is determined by the current app page $s.p$ and its corresponding execution environment $s.env$, denoted as $s \uparrow (p, env)$; ❷ Each event e must has a widget w in the page p with operable attributes $p(w).attrs$ (detailed in §IV-C) as its carrier, and the e has to meet the $p(w).attrs$, denoted as $e \rightsquigarrow p(w).attrs$.

IV. EP-Detector

Given a target app, EP-Detector identifies EPAs for each widget, through the following three modules, as shown in Figure 2. The **Target Navigation** module navigates to the target pages and widgets relying on the *page & widget identification*. To reduce cost, the *Event Trace* during the navigation is logged in the **Recorder** to guide new navigation. The *target pages and widgets* are then fed into the **Detection Execution** module to simulate the execution paths triggered by the EPA-sensitive events (defined in §IV-C). Finally, the **Test Oracle** module incorporates an automated oracle to compute the $Diff$ function (Eq. (3)), determining the existence of three types of EPAs. In this process, *change of environment* Env_{sim} and *page similarity* $\mathcal{P}_{sim}$ before and after an event are computed. The system environment is collected by the **Resource Monitor**, and $\mathcal{P}_{sim}$ is used to determine if a target page is reached in Target Navigation and Detection Execution.

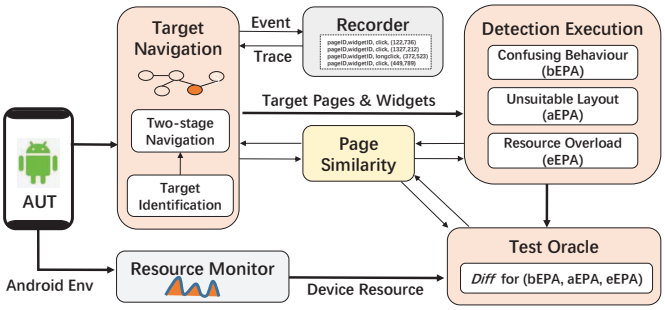


Fig. 2. The Workflow of EP-Detector.

A. Two-stage Navigation

To identify and locate widgets (e.g., buttons, text editors, sliders) that may cause error-prone operations, EP-Detector adopts a *widget-exploration* approach, deviating from the typical path-exploration approach in the majority of existing model-based GUI testing [19]–[23]. In this process, the Target Navigation module incorporates two stages.

Stage 1: Page Navigation. Let $s.p$ be the original page where s is the execution state of an EPA detection, the target page $s_t.p_t$ can be reached in a sequence of state transitions $\langle \delta(s, e_{1_1}) \rightarrow s_{1_1}, ..., \delta(s_{1_i}, e_{1_i}) \rightarrow s, ..., \delta(s, e_{k_1}) \rightarrow s_{k_1}, ..., \delta(s_{k_{j-1}}, e_{k_j}) \rightarrow s_t \rangle$, named as the *Event Trace*. Essentially, we try to trigger each element in the original page in the *Breath First Search* (BFS) way following the order of the page layout. Each triggering of an element v leads to various states $(\langle \delta(s, e_{v_1}) \rightarrow s_{v_1}, ..., \delta(s_{v_i}, e_{v_i}) \rightarrow s \rangle)$ with a state transition back to the original state s at the end. The event trace is the concatenation of multiple element triggering transactions until the target event is identified.

Stage 2: Widget Navigation. Once the target page p is located, we proceed to identify the target widget $p(w)$ by matching its index. The widget is characterised by its error-prone attributes $p(w).attrs$, which is used to initiate the error-prone operations in the subsequent EPA detection by triggering the corresponding event $e \rightsquigarrow p(w).attrs$.

To reduce exploration overhead, optimization strategies are implemented based on the following two observations.

Observation 1. *Reversing to the previous state occupies a great majority of page navigation tasks.*

As shown in Stage 1, detecting the EPAs on a target widget $p(w)$ at state s requires triggering multiple error-prone events at the page $s.p$ forming an event trace. For

each triggering of an element v with transition $\langle \delta(s, e_{v_1}) \rightarrow s_{v_1}, ..., \delta(s_{v_i}, e_{v_i}) \rightarrow s \rangle$, EP-Detector ❶ collects detection element information of s_{v_i} for later steps; ❷ navigates to the original s for next transition. One can observe that for a given s there are many reverse transitions $\delta(s_{v_i}, e_{v_i}) \rightarrow s$. This observation inspires the *Optimization 1*.

Optimization 1. Reverse transition $\delta(s_v, e) \rightarrow s$ is implemented (by clicking the page-level back buttons) in the EPA detection with the highest priority in page navigation.

Observation 2. *Not all state transitions lead to new page rendering when execution anomaly occurs.*

If a system crash, app freeze, or failed rollback ocurs, the reverse transition in *Optimization 1* becomes invalid. To ensure the page navigation still restart from the initial state s, we record the navigation trace, which is a sequence of event-widget pairs $(e_i, p_i(w_i))$ where e_i is a triggering event and $p_i(w_i)$ is the corresponding widget w_i in page p_i. The idea is to leverage a *space-for-time* strategy for an express navigation to the target page, ignoring the repeated page exploration.

Optimization 2. Each navigation trace for a target page p_t, i.e., $\langle (e_1, p_1(w_1)), ..., (e_i, p_i(w_i)), ..., (e_t, p_t(w_t)) \rangle$ is recorded in the **Recorder** module to guide subsequent repeated navigation from the initial state.

B. Target Identification

The process of page navigation and widget navigation involve frequent decision-making to determine whether the target page or widget is reached, defined in *page identification* and *widget identification* respectively.

1) Page Identification: In traditional model-based dynamic testing, the page/state identification often relies on *absolute-matching*. That is, the attributes of the target page/state is first encoded using abstraction techniques e.g., hash code [19] or customized template [20], [22]. Then a given page/state is identified by matching the abstraction results. However, such absolute-matching is often invalid in detecting real-world apps.

Observation 3. *The attributes of the target page may change if being repeatedly accessed. Formally, let s be an original state, s_t and s'_t be two target states transited from s with the same event trace $\langle e_1, e_2, ..., e_k \rangle$, we may have $s_t.p.attrs! \equiv s'_t.p.attrs$, where $\equiv$ denotes the identical relation and $p.attrs$ denotes the attributes of the page p.*

This phenomenon is often observed in *deep-interactive* apps, e.g, `AcFun`, `Bilibili` and `Loklok`, and its root-cause lies in the updating of *dynamic elements*, e.g., advertisements, random animations, or *page reconstruction*. In details, page updating during each rendering may: ❶ Increase/reduce *attributes for localization*, e.g., `<elementId>`, `<resourceId>` and `<class>`; ❷ Modify the *attributes for content and action*, e.g., `<text>`, `<isEnable>` and `<clickable>`. Therefore, traditional attribute-based absolute-matching may fail to identify pages, leading to decreased precision of navigation. This inspires us to explore an *relative-matching* identification by introducing a *Jaccard Distance* based *page similarity*:

$$\mathcal{P}_{sim}(p_t, p'_t) = \frac{\#(p_t.attrs \cap p'_t.attrs)}{\#(p_t.attrs \cup p'_t.attrs)}. \tag{2}$$

If the $\mathcal{P}_{sim}$ is larger than a given threshold τ_p, the target page is identified. In practice, not all the page attributes are needed in the computation. We prefer attributes for *localization* rather than for content and action, due to their high stability during the page reconstruction. The utilization of $\mathcal{P}_{sim}$ spans across multiple stages of EP-Detector:

- In the *page navigation* (§IV-A), the $\mathcal{P}_{sim}$ is used to identify the target page. This similarity-based navigation is also frequently used in the computation of *Diff*.
- In the *test oracle* stage (§IV-D), the $\mathcal{P}_{sim}$ plays a critical role in computing the *Diff* function (Eq. (3)) that is relied upon to automatically determine an EPA in the test oracle. The function *Diff* calculates the differences between two given states s and s_t by comparing the similarity of both their page and environment, as they collectively define an execution state (i.e., $s \uparrow (p, env)$ in §III). Therefore *Diff* is computed as:

$$\begin{aligned} \mathit{Diff}(s, s_t) = & 1 - (\alpha \times \mathcal{P}_{sim}(s.p, s_t.p) \\ & + \beta \times Env_{sim}(s.env, s_t.env)), \end{aligned} \tag{3}$$

  where $Env_{sim}(s.env, s_t.env)$ denotes the environment similarity, defined in the following Eq. (4).

$$Env_{sim}(n1, n2) = \sum_{env_i \in EnvSet} Z_i \ \Delta_{env_i}(n1, n2), \tag{4}$$

  where $EnvSet$ is the set of environment factors, e.g., CPU, RAM and network; Z_i is the normalization coefficient; Δ computes the value change between $n1$, $n2$.
- In the *detection execution* stage (§IV-C), the $\mathcal{P}_{sim}$ also serves as *state abstraction* in EPA detection. Specifically, pages exhibiting similarity within a predefined threshold are treated as identical, and states sharing identical pages are clustered together as the same *page state*.

2) Widget Identification: In the two-stage target navigation, once the target page is identified, the target widget can be identified based on the types of widgets that are frequently interacted with users, e.g., `Button`, `Checkbox` and `Scrollbar`. However, such type-based identification may fail in practical detection.

Observation 4. *Not all interactive widgets are capable of executing the expected operations, necessitating their detection.*

For example, a general `ImageButton`'s reaction to *long click* depends on the `long-clickable` attribute setting, rather than the `Button` type. Under this insight, the *widget identification* of EP-Detector is *attribute-based* rather than type-based. We consider a concise but effective attribute set for *widget identification* specific for the EPA operations (listed in Table I). The identification procedure contains four steps: ❶ Filter the widgets through the *Status* attributes; ❷ Determine the operation behaviours through *Action* attributes (Note that the *Action* attributes only cover EPA-related behaviours); ❸

TABLE I
EXTRACTED WIDGET ATTRIBUTES.

Type	Name	DType	Type	Name	DType
ID	element-id	text	Links	href	text
Action	clickable	bool	Status	visibility	category
	scrollable	bool		displayed	bool
	checkable	bool		enabled	bool
	longclickable	bool	Size	bounds	numeric

Determine the operation area through *Size* attribute; ❹ Record the *ID* for further check and comparison. Based on these attributes, user behaviours under specific scenarios are simulated and realized (see §IV-C1).

C. Detection Execution

The occurrence of an EPA requires the integration of three core factors, i.e., *subject*, *object* and *environment* as stated in §I. Under this philosophy, the EPA can be partitioned into three types according to these factors, i.e., *confusing behaviours* (bEPA), *unsuitable layout* (aEPA) and *extreme resources* (eEPA), detailed in each subsection.

1) Confusing Behaviours: The types of user behaviours working on widgets are complicated, making it challenging to determine the confusing ones. Particularly it is difficult to consider behaviors that may be interfered with by natural factors such as voice control, voice input and phone shaking, due to their high uncertainty. Therefore, our focus is shifted to the user gesture events. Guided by the *gesture callbacks* contained within gesture-related Android classes (e.g., `android.view.View` [24], `android.widget` [25], `android.view.GestureDetector` [26] and `MotionEvent` [27]), we can summarize the user behaviours associated with these callbacks. However, detecting all behaviours for each widget is costly. Redundant and less critical behaviours are filtered using the following strategies.

- For the behaviours with multiple directions, only the most commonly used one is preserved, e.g., we preserve *upSwipe* and filter out *downSwipe*.
- For similar behaviours, only the representative one is preserved, e.g, we preserve *swipe* and filter out *fling*.
- The system-specific and rarely used behaviours, e.g., *knuckle tap* and *twoFingerDoubleClick*, are omitted.

As a result, 12 typical behaviours are collected, shown in Table II, where the behaviour name starts with the number "2" represents the behaviour has two operation directions, and behaviours used with low-frequency in experimental trials are denoted with "*", appearing only in specific apps.

TABLE II
CONFUSING GROUPS OF COLLECTED BEHAVIOURS.

Category	Gesture	Widget Attributes
CLICK	click, doubleClick, twoFingerClick*, longClick, swipe	clickable, checkable
LONGCLICK	longClick, click, swipe, twoFingerLongClick*	long-clickable
SCROLL	2scroll, 2swipe, 2twoFingerSwipe*, pinchOpen*	draggable, scrollable

The behaviours are divided into three confusing groups: `CLICK`, `LONGCLICK` and `SCROLL` based on their *confusion degree*, defined as follows:

Definition 1. *Confusion Degree. Given two user behaviours h_a and h_b, their confusion degree is the sum of their distances on the four operation dimensions—the number of operations opN, number of fingers opF, operation distance opD and duration opT, i.e., $conf(h_a, h_b) = \sum_{opX \in OP} |h_a.opX - h_b.opX|$, where $OP = \{opN, opF, opD, opT\}$.*

The values of opN and opF are obtained by a simple counting of the behaviour. opT and opD are discretized coarsely based on their semantics. For opT, "standard" and "long" are quantified as 1 and 2; for opD, only "in-place operations" and "operations with distance" are distinguished, quantified as 0 and 1 respectively. For example, the $\{opN, opF, opD, opT\}$ of *doubleClick* and *swipe* are $\{2, 1, 2, 0\}$ and $\{1, 1, 2, 1\}$ respectively, and their confusion degree can be computed as $|2-1| + |1-1| + |1-2| + |0-1| = 2$. The *confusing behaviours* are identified based on their pairwise *confusion degree*. Each group has a *basic behaviour* (underlined in Table II) h; additional behaviours h_x in the same group are identified if the equation $conf(h, h_x) <= 2$ holds.

After that, the confusing behaviours in the same group are simulated one by one, whose results are recorded for the determination of bEPAs by the following Test Oracle. The basic behaviour simulation is implemented by calling APIs in `appium`. But there are no available APIs for operations like *doubleClick* and *twoFingerSwipe*; they are achieved by simulating the combination of basic behaviours.

2) Unsuitable Layout: Unsuitable widget design prone to EPAs encompass a range of factors [28], including colour, shape, size, spacing and textual states [29]. However, only layout-related factors, e.g., *size* and *spacing*, can be checked through dynamic detection, which are the primary focus of EP-Detector. EPAs triggered by *unsuitable layout* is called *aEPAs.*

Definition 2. *Unsuitable Layout. Given the center point p_c of the target widget w_t, the minimum width $wid_{min}(w_t)$ of w_t is defined as $min\{Dis(p_c, p_x)|p_x \in P_t\}$, where P_t is the set of boundary points of w_t. Assuming that w_n is another widget closest to w_t, the minimum spacing distance $spc_{min}(w_t, w_n)$ between w_t and w_n is defined as $min\{Dis(p_c, p_n)|p_n \in P_n\}$, where P_n is the set of boundary points of w_n. An unsuitable layout is the widget design that satisfies both $cond_1$ and $cond_2$, where $cond_1 : wid_{min}(w_t) < \tau_w$ violates the specified safe distance for widget size τ_w, and $cond_2 : spc_{min}(w_t, w_n) < \tau_s$ violates the safe space between widgets τ_s.*

Figure 3 illustrates typical cases of unsuitable layout: (a) refers to a suitable widget size (i.e., $!cond_1$); (b) indicates an unsuitable size but suitable spacing (i.e., $cond_1$ & $!cond_2$); (c) indicates an unsuitable layout (i.e., $cond_1$ & $cond_2$). From this figure, we can also see that we only need to use τ_w in defining the *safe area*, as τ_s serves the purpose.

Definition 3. *Safe Area. Given a center point p_c of the target widget w_t, the safe area for w_t is a circular area with the center being p_c and the radius being τ_w.*

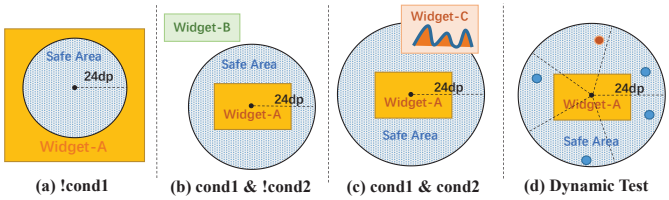


Fig. 3. Cases of Unsuitable Layout.

The τ_w can be determined through the guidance of Android suggestion for accessible user experience [30], where they *recommend that each interactive UI element should have a focusable area, or touch target size, of at least* ***48dpx48dp***. Thus, the τ_w is set to the same size value accordingly.

The *unsuitable design* can be detected statically during the page navigation, since the boundary of the target widget can be captured through parsing the layout *.xml* tree. However, in static detection, we could not observe the aEPAs and their properties like location. Hence, dynamic operation simulation is used in EP-Detector. As shown in Figure 3(d), we could guide the dynamic simulation of operations to approach the boundary of *safe area* and evenly distributed in the *safe area*, to avoid the location concentration of simulated operations.

3) Resource Overload: Insufficient device resources for app running, e.g., CPU, RAM and network capacity, can result in EPAs. For instance, when the device resources are overloaded and about being exhausted, *execution anomaly* such as page freezing and operation failing, are prone to occur, interrupting user operations and leading to misoperations. In response to execution anomalies, users commonly resort to *continuous-and-repeated (c&r)* operations, despite their high-risk nature. This practice results in two types of eEPAs shown in Figure 4.

Heavier Overload. As indicated in Figure 4(a), the *c&r* operations on the same widget of a frozen page would incur *heavier overload* on running resources, e.g., CPU, RAM and network, which can raise further irreversible running exceptions, e.g., system crash and data loss.

Unintended Misoperation. As indicated in Figure 4(b), the *c&r* operations are first performed on the frozen `page A`. When the page suddenly jumps, it is difficult for a user to realize in time that the user may mistakenly apply the same *c&r* operations to the same position on the new `page B`, resulting in an unintended exception.

The two types of eEPAs that triggered by real running environment are hard to be checked by static analysis. So EP-Detector employs a dynamic analysis to detect them, which includes three steps. ❶ Try to *click* the widgets with `clickable=true` on `page A`, and checks whether `page A` jumps to another `page B`. ❷ If no jump occurs, continuously simulate identical *click* operations and performs them at the same position of `page A` for 5 times, with the interval

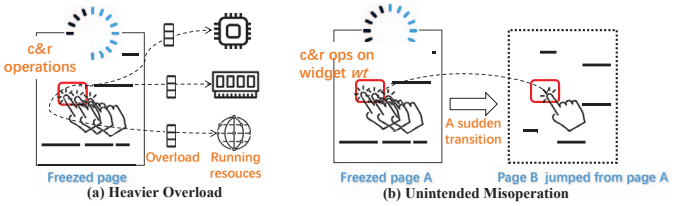


Fig. 4. Two Types of eEPAs.

of 0.3s (the value of times and interval correspond to the average *c&r* frequency and speed of humans, respectively [31]); simultaneously monitoring the changes of CPU, RAM and network speed. ❸ If a jump occurs, simulate the *click* operation at the same position of `page B` and monitors whether a new state transition happens. Note that, for the sake of cost-efficiency, EP-Detector only supports *click* simulation in the detection. In implementation, EP-Detector uses `Thread.sleep()` to suspend the `UI main thread` to simulate the page freezing.

D. Test Oracle

EP-Detector supports an automated test oracle, where the function *Diff* (Eq. (3) in §III and §IV-B) is computed using different strategies for the three types of EPAs.

Detection of *bEPA*. Given current state s and a pair of *confusing behaviours* (e_1 and e_2) from the same behaviour group in Table II, the *test oracle* aims to determine whether $(Diff(s, \delta(s, e_2))) > \tau \wedge Diff(\delta(s, e_1), \delta(s, e_2)) > \tau$ holds. If it does, a bEPA is found. The underlying rationale is that once the new page triggered by the confusing behaviour (e_2) is neither the same as the original page (s) nor the same as the page triggered by the basic behavior (e_1), it could lead to undesirable consequences.

Detection of *aEPA*. Given current state s and a pair of identical operations on *safe area* (§IV-C2) (e_1 and e_2), the *test oracle* aims to determine whether $Diff(\delta(s, e_1), \delta(s, e_2)) > \tau$ holds. If it does, an aEPA is found. The underlying principle is that two identical operations within the *safe area* should lead to the same outcomes; otherwise, it could result in anomalies.

Detection of *eEPA*. The test oracle contains two parts: ❶ *Heavier overload.* Assuming s and s' are the states before and after *c&r* operations respectively, the oracle determines whether $Env_{sim}(s.env, s'.env) < \tau_{env}$ holds; ❷ *Unintended misoperation.* Through *page identification*, the test oracle first checks if a *page jump* from `page A` to `page B` occurs when *c&r* operations are performed. Then assuming s and s' are the states before and after operating on `page B` respectively, the oracle determines whether $Diff(s, s') > \tau$ holds. If both conditions are satisfied, an eEPA is found. The principle here is that an unintended page jumping accompanying with a heavier overload causing significant environment changes may result in risky consequences.

Determining α and β. To calculate the *Diff* function, in addition to computing $\mathcal{P}_{sim}$ and Env_{sim} following their formula Eq (2) and Eq (4), we need to determine the parameters α and

β. Except for the detection of *heavier overload*, the parameters α and β in *Diff* formula (see Eq. (3)) are determined in a dynamic way as follows.

$$\{\alpha, \beta\} = \begin{cases} \{1, 0\}, & \mathcal{P}_{sim} < \tau_p \\ \{0, 1\}, & \mathcal{P}_{sim} > \tau_p \ \& \ Env_{sim} < \tau_{env} \\ \{0.5, 0.5\}, & otherwise. \end{cases} \tag{5}$$

V. Implementation

EP-Detector is implemented in Python 3.8 with more than $4k$ lines of code excluding library files. It adopts existing API support libraries, including the Appium dynamic test API, the `ADB` screenshot command, the `cv2` command for drawing points, the `xml.etree` plugin for parsing *xml* files, the subprocess pipeline command for sending `ADB` commands, etc. Specifically, to meet the detection request of *bEPA*, extra user behaviours are simulated and customized based on the Appium, including *drag-and-drop* operations and *multi-touch* interactions. The experiments are conducted on multiple versions of Android (i.e., Android 11.0, 12.0 and 13.0). The test device is equipped with 8`GB` RAM and 8 cores CPU.

Following the approach in §IV, for each app target, we first navigate to each widget to trigger all designed simulated events in each type of EPA, and uses the test oracle to determine the EPAs. To decide the best values for the parameters—τ_p (Page Identification), τ and τ_{env} (Test Oracle), a common-used parameter optimization method—*grid search*, was employed to fine-tune these parameters with a step of 0.1 within an imperial value range $0.1 - 0.5$ for τ and τ_{env} and $0.5 - 0.9$ for τ_p. Optimal performance in EPA detection was observed when $\tau_p = 0.8$, $\tau = 0.2$ and $\tau_{env} = 0.3$, where 92.0% of the EPAs and 76.7% page states were accurately identified on 30 testing pages.

VI. Evaluation

Based on the results of the above execution, the EP-Detector is then evaluated from four perspectives: *precision*, *completeness*, *efficiency* and *severity* of the detected EPAs.

A. Precision

1) Benchmark: To ensure the fairness and effectiveness of the evaluation, we select the evaluation benchmark with the following three rules. ❶ Comparability: Collect the three apps (in Loc) evaluated by the existing dynamic GUI testing works [20] [32], i.e., `WordPress`, `K-9 Mail`, and `MyExpense`. ❷ Prevalence: Collect the top apps with download amounts of no less than $100k$. ❸ Comprehensiveness: To balance the coverage of both categories and complexity, the collected apps are evenly distributed across 9 functional categories; the number of pages in the apps is evenly distributed across the intervals $[0, 10]$, $[10, 30]$ and $[30, \infty]$; the apps should be collected from multiple markets. Following these rules, we collect 5 popular apps for each category in Google Play, and 5 top apps in Chinese Market, i.e., `WeChat`, `AliPay`, `Bankcomm`, `Bilibili` and `UC Browser`. At last, 53 eligible real-world apps are collected as the benchmark.

2) EPA Confirmation: The detection outcomes of EP-Detector are automatically marked EPAs on the anomalous widgets of GUI screenshots, which need to be manually verified. We recruited 3 volunteers with rich experience in using mobile systems in our college to conduct the confirmation. To ensure the fairness of the confirmation, we conducted necessary training for the volunteers, which was mainly divided into three stages: ❶ Familiarize the volunteers with the process of target navigation and EPA detection, especially with the detection indicators introduced in test oracle (§IV-D). ❷ Randomly select 60 existing EPA samples (20 for each type of EPA) for them to identify. If one sample is identified incorrectly, an additional 5 samples are added to identify until they are all identified correctly. ❸ Ensure that each detected EPA and navigated page are confirmed by at least two volunteers. If there is contradiction, the third volunteer or the author completes the final confirmation. In total, the manual confirmation for both page states and detected EPAs takes $23 * 8$ hours.

3) Results: The apps with detected EPAs in the benchmark, along with their precision computed after confirmation, are presented in Table III. The overall precision (88.28%) is encouraging, which indicates the EP-Detector can significantly assist in saving on manual inspection efforts in detecting the EPAs. Among the three types of EPAs, the tool has the highest accuracy in detecting bEPAs (91.52%), and the lowest accuracy for eEPAs (85.81%). This is mainly because the occurrence of eEPAs requires specific runtime environment, and it is difficult to ensure the stability of the system environment consistently during app operation. In contrast, the detection of bEPAs focuses only on a few operations in specific positions on the page, reducing the likelihood of confused operations during the detection. To delve deeper into the limitations of EP-Detector in the detection precision, we randomly select 100 (about 20%) of the false positive cases and analyze their underlying causes, which are summarized as follows:

- Nearly half of the false detected cases ($47/100$) are caused by the *intersecting influences* among specific system resources or execution processes. For example, the continuous use of network resources by streaming media on the target page may lead to confusion in resource consumption when detecting EPAs on other widgets on the page, resulting in falsely reported anomaly.
- Nearly a third of the false cases ($34/100$) result from settings for resources (e.g., RAM and network traffic) and similarity parameters (e.g., τ and τ_{env} in §IV-C). Variations in widget functionalities lead to different degrees of resource consumption. Setting a low threshold may cause EP-Detector to falsely label healthy widgets as error-prone (*false positives*), while a high threshold may result in overlooking EPAs (*false negatives*).
- The remaining nearly $1/6$ false cases are mainly raised from extraneous factors such as interruptions of system-level events and abnormal fluctuations in network.

We would like to highlight that these false positives do not

TABLE III
PRECISION PERFORMANCE OF EP-DETECTOR IN EPAS DETECTION.

App	Size (MB)	Download	% Precision of Detected EPAs			
			aEPA	bEPA	eEPA	Total
iTranslate 5.7.2	114.8	50M+	100 (3/3)	86.49 (32/37)	89.83 (53/59)	88.89 (88/99)
Battery HD Pro 1.99.15	17.5	100K+	100 (3/3)	95.83 (23/24)	97.87 (46/47)	97.3 (72/74)
Memo 2.9.7	6.41	10M+	0 (0/4)	88.46 (23/26)	86.21 (25/29)	81.36 (48/59)
Notepad 1.33.1	6.6	10M+	71.43 (5/7)	90 (18/20)	50 (1/2)	82.76 (24/29)
Car Launcher 3.4.1.24	24.8	1M+	50 (2/4)	87.5 (14/16)	69.23 (9/13)	75.76 (25/33)
Bankcomm 8.0.0	186.7	50M+	79.69 (51/64)	98.25 (56/57)	75.44 (43/57)	84.27 (150/178)
Days Matter 1.18.19	30.5	1M+	100 (1/1)	100 (17/17)	85.71 (18/21)	92.31 (36/39)
Alipay 10.5.76	136.8	10M+	85.6 (143/167)	88.89 (80/90)	84.38 (27/32)	86.51 (250/289)
Everydoggy 1.71.2	30.8	500K+	56.52 (13/23)	84 (21/25)	54.05 (20/37)	63.53 (54/85)
Frandroid 6.0.13	27.8	500K+	100 (13/13)	100 (7/7)	100 (0/0)	100 (20/20)
QR 2.2.58	9.8	500M+	100 (2/2)	66.67 (2/3)	100 (1/1)	83.33 (5/6)
GJJ 3.12.0	33.7	100K+	78.26 (18/23)	100 (171/171)	88.36 (167/189)	92.95 (356/383)
Right Gallery 5.0.4	34	100K+	100 (8/8)	95.83 (23/24)	100 (0/0)	96.88 (31/32)
Google Books 193791	19.2	100M+	88.24 (30/34)	66.67 (6/9)	72.73 (24/33)	78.95 (60/76)
Google Maps 11.120	180.4	10B+	100 (19/19)	95 (19/20)	93.33 (14/15)	96.3 (52/54)
Google Translate 8.4.75	33.5	1B+	100 (25/25)	88.46 (23/26)	89.47 (17/19)	92.86 (65/70)
Gmail 614808802	137	10B+	100 (9/9)	100 (29/29)	100 (2/2)	100 (40/40)
Google Earth 10.46.0.2	49.1	500M+	91.67 (22/24)	80.56 (29/36)	64.29 (9/14)	81.08 (60/74)
GCamator 5.1.17	14.3	10M+	95.24 (20/21)	98.15 (53/54)	66.67 (10/15)	92.22 (83/90)
Listen Free 1.8.7	34.8	100K+	66.67 (8/12)	87 (87/100)	91.15 (103/113)	88 (198/225)
IQilu 0.2.23	94.2	100K+	100 (24/24)	64.29 (45/70)	85.53 (260/304)	82.66 (329/398)
Jetour Traveller 3.2.9	212.6	1M+	33.33 (8/24)	94.69 (107/113)	85.63 (143/167)	84.87 (258/304)
Komorebi Memo 4.5	23.4	5M+	100 (8/8)	100 (8/8)	100 (5/5)	100 (21/21)
Photo Editor Pro 6.7.5.1	70.9	10M+	100 (1/1)	96.77 (60/62)	96.25 (77/80)	96.5 (138/143)
LINE Webtoon 3.2.1	39.1	100M+	89.36 (42/47)	76.92 (20/26)	70 (7/10)	83.13 (69/83)
Car Scanner 1.105.1	77.3	10M+	33.33 (1/3)	85.19 (23/27)	76.92 (10/13)	79.07 (34/43)
PicsArt Studio 1.0.2	25.8	1B+	100 (9/9)	91.3 (84/92)	85.61 (119/139)	88.33 (212/240)
Pleco 3.2.93	135.3	5M+	82.86 (29/35)	87.72 (50/57)	55.17 (16/29)	78.51 (95/121)
Music Speed 12.0.0b5	17.9	10M+	50 (1/2)	66.67 (6/9)	80 (4/5)	68.75 (11/16)
News 9.5.8	76.7	900M+	94.12 (16/17)	84.21 (16/19)	66.67 (4/6)	85.71 (36/42)
Mangasuki 1.5d	8.1	100K+	75 (3/4)	91.18 (31/34)	96.67 (29/30)	92.65 (63/68)
WeChat 8.0.47	252.4	100M+	90.2 (138/153)	95.19 (178/187)	95.1 (194/204)	93.75 (510/544)
China Daily 8.0.8	14.2	1M+	100 (3/3)	97.56 (40/41)	100 (45/45)	98.88 (88/89)
Skyeye 13.12.10	40.3	100M+	100 (1/1)	100 (11/11)	92.59 (25/27)	94.87 (37/39)
UC Browser 16.2.1	100.5	800M+	96.08 (49/51)	96.97 (32/33)	100 (0/0)	96.43 (81/84)
Weather 5.4.4	46.03	1M+	50 (1/2)	66.67 (2/3)	100 (3/3)	75 (6/8)
Lazycook 3.0.0	12.6	50M+	93.88 (46/49)	100 (69/69)	88.31 (68/77)	93.85 (183/195)
Zillow Map 15.6.0	180.6	10M+	89.47 (17/19)	62.5 (5/8)	37.5 (3/8)	71.43 (25/35)
FaceApp 11.9.3	83.3	500M+	100 (1/1)	100 (18/18)	94.44 (17/18)	97.3 (36/37)
Raise to Answer 3.6.5	2	100K+	100 (0/0)	90.91 (10/11)	100 (0/0)	90.91 (10/11)
MultiNotes 2.87	38.9	5M+	85.71 (6/7)	40 (4/10)	33.33 (1/3)	55 (11/20)
LanDroid 1.43	0.2	500K+	91.67 (11/12)	90 (36/40)	67.74 (21/31)	81.93 (68/83)
SocksDroid 1.0.3	0.8	500K+	66.67 (2/3)	100 (26/26)	0 (0/1)	93.33 (28/30)
AnyMemo 10.11.7	4.6	100K+	95 (19/20)	92.31 (12/13)	100 (4/4)	94.59 (35/37)
Lumii 1.630.149	32	50M+	100 (2/2)	88.24 (15/17)	94.74 (18/19)	92.11 (35/38)
Motolog 3.20.9	11.2	100K+	91.67 (11/12)	94.44 (17/18)	50 (2/4)	88.24 (30/34)
ZArchiver 1.0.9	4.9	100M+	100 (5/5)	94.74 (72/76)	100 (2/2)	95.18 (79/83)
TuneIn 33.6.3	84.5	100M+	85.71 (18/21)	93.1 (54/58)	69.23 (9/13)	88.04 (81/92)
Loklok 1.7.1	44.1	5M+	100 (5/5)	94.2 (130/138)	84.71 (144/170)	89.14 (279/313)
WordPress 24.4	167.4	10M+	96.97 (32/33)	95.08 (58/61)	88.46 (46/52)	93.15 (136/146)
K-9 Mail 6.801	9.6	5M+	100 (13/13)	90.48 (38/42)	100 (0/0)	92.73 (51/55)
MyExpense 3.3.7	10.79	1M+	88.46 (46/52)	96.36 (53/55)	92.68 (38/41)	92.57 (137/148)
Bilibili 3.16.0	171.5	800K+	77.97 (46/59)	77.14 (54/70)	79.85 (107/134)	78.71 (207/263)
Average	60.8	492.9M+	86.74 (19/22)	91.52 (40/44)	85.81 (38/44)	88.28 (97/110)

necessarily indicate significant precision flaws in EP-Detector. On the one hand, the intersecting influences and extraneous factors are difficult to completely eliminated during app execution. On the other hand, the trade-off raised by parameters belongs to the inherent challenge in the bug detection tasks, and we have mitigated the performance degradation through various techniques, such as grid search in §VI.

Summary: EP-Detector shows a high effectiveness in EPA detection with an average precision of 88.28%. The factors that constrain the precision include *intersecting influences*, *parameter settings* and *work-extraneous impacts*.

B. Completeness

The completeness is evaluated from two perspectives: the *coverage of page state* and the detection *false positive*.

1) Coverage of Page State: To evaluate the coverage of page state, we compared EP-Detector with two widely-used dynamic GUI testing tools, Stoat and APE. Stoat [19] simulates user-device interactions by constructing a random finite state machine model of the apps with both static and dynamic analysis. APE [20] is a dynamic GUI testing framework employing decision tree to represent dynamic page states.

Metric. The *coverage of page state* is measured by the ratio of the explored page states to the total ones. Note that to alleviate navigation redundancy, the page states with $\mathcal{P}sim > \tau$ are treated as the same page state.

Benchmark. When Stoat and APE were used to test the benchmarks, some apps frequently crashed or stopped midway (might be raised by the incompatibility to the Android version). To ensure fair comparison, we conducted an in-depth analysis of 10 apps that are fully tested on both frameworks.

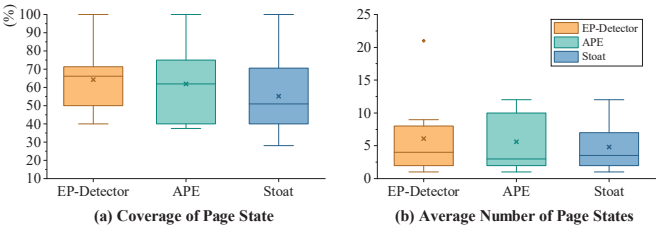


Fig. 5. Comparison of Page Coverage.

Results. As illustrated in Figure 5 (a), the coverage performance of EP-Detector is slightly better than the other two methods, which achieves an average coverage rate of 60.40%, surpassing APE's 55.45% and Stoat's 47.52%. Through manual confirmation, there are 101 valid page states in total (detailed in Table III), where EP-Detector detects 61 of them, averaging 6.1 page states per app (illustrated in Figure 5 (b)). To ensure the effectiveness of the simulated operations, EP-Detector neither employs tree-based abstraction like APE nor uses a dynamic-static hybrid method like Stoat. Instead, EP-Detector employs a novel approach through comprehensive widget navigation and fine-grained similarity calculations to achieve promising page coverage during the simulation of error-prone operations. We also carefully analyzed the page states that were not navigated, primarily due to two reasons: First, some page states require to be triggered by events that are difficult to simulate, such as password authentication and *QR code* input. Second, dynamic widgets (e.g., ad widgets) that vary in each round of visiting significantly disturb the page navigation, occasionally making target pages unreachable.

Summary: EP-Detector achieves an average coverage rate of 60.40%, surpassing APE's 55.45% and Stoat's 47.52%. The failed page navigation mainly attributes to *specific events* and *dynamic widgets*.

2) False Negative: Due to the absence of dedicated detection work and complete ground truth for EPAs, directly identifying false negatives of the EP-Detector outcomes is challenging. To alleviate this evaluation challenge, we investigate a widely used industry product—*Google Accessibility Scanner* [33], which partially overlaps with EP-Detector in detecting EPAs. The Accessibility Scanner evaluates widgets from three perspectives: color contrast, labeling for guidance and touch size.

The detection results for the benchmark of 10 apps is depicted in Figure 6. EP-Detector identified 417 confirmed EPAs, whereas Accessibility Scanner detected 362 anomalies, with 156 aEPAs being identified by both tools. Unlike EP-Detector,

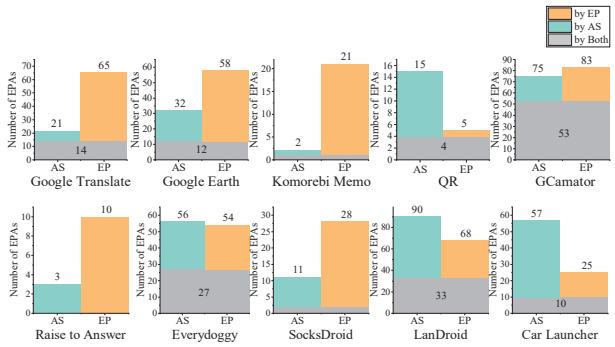


Fig. 6. Result Comparison between Accessibility Scanner and EP-Detector.

Accessibility Scanner detects EPAs solely through static analysis of widget attributes (e.g., `android:background`, `android:layout` and `android:text`). For that reason, it hardly detects the bEPAs (135 confirmed) and eEPAs (93 confirmed) which requires dynamic operation simulation. As to the area-related anomalies, the Accessibility Scanner reports significantly more results (362 vs. 189) than EP-Detector, especially for the typical two apps `Car Launcher` and `QR`. Except for the anomalies raised by the color contrast and label hint, the Accessibility Scanner checks 174 touch size-related anomalies, where 18 of them cannot be checked by EP-Detector. From further analysis, the touch sizes of 14 (out of the 18) detected widgets are indeed smaller than the standard, but there are no other widgets within their *safe areas*, which are unlikely to cause error operations. The remaining 4 anomalies are identified as the *false negatives* of EP-Detector. Through careful analysis, the overlooked EPAs are raised by the invalid simulated operations under the slow network and laggy system.

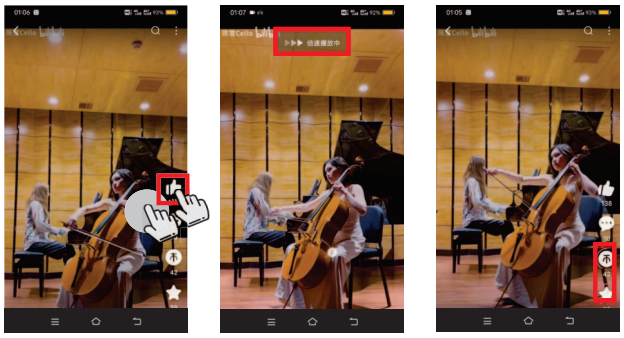
Summary: EP-Detector reports significantly more confirmed EPAs than Accessibility Scanner in detecting 10 typical apps, but still have 4 aEPA false negatives due to invalid simulation.

C. Efficiency

To assess the efficiency of EP-Detector, we have recorded the time consumption details during the detection of 53 apps, illustrated in Table IV. On average, each app costs about 6.3 hours, which is acceptable for offline testing compared to existing dynamic testing works [20] [32]. The efficiency is significantly influenced by the number of pages and widgets. For example, `WeChat` spends the longest detection time with 19.4 hours, attributed to its substantial 108 page states and 1604 widgets. For detailed analysis, among the modules and stages of EP-Detector, the *target navigation* is the most time-consuming module, accounting for 65.1% of the total time. This is primarily due to the BFS-based navigation, which initiates numerous restarts from the original page (§IV-A). Despite optimizations implemented including simulating reverse transition and building trace recorder, the time consumption for this part remains substantial.

TABLE IV
AVERAGE EFFICIENCY OF EP-DETECTOR IN EPA DETECTION.

# App(hr)	# Page(min)	# Widgets(min)	# Navigation(hr)	Oracle(hr)	# Exec(hr)
6.3	24.8	1.1	4.1	1.3	0.9



(a) Positions of O2T and Video Acceleration Widgets. **(b) Animation Prompt by Video Acceleration Event.** **(c) Animation Prompt by O2T Event.**

Fig. 7. aEPAs Caused by *One-click Triple-interaction* and Video Acceleration.

Summary: The efficiency of the EP-Detector is overall acceptable for offline detection, with 6.3 hours per app and 24.8 minutes per page. However, the time overhead of specific apps may be significant due to complex app design.

D. Severity

From Table III, EP-Detector on average detects 96.91 confirmed EPAs for each app, with *one EPA identified for every two widgets*. The aEPAs are the least prevalent in the apps, with only 19.04 per app and 1.87 per page, while bEPAs are the most common ones, with an average of 39.94 per app and 3.91 per page. Apps with simple page layouts, like `QR`, `Weather`, typically have few (less than 10) EPAs, which aligns with intuitive expectations. On the contrary, apps like `Alipay` and `WeChat` have a high prevalence of EPAs (more than 200), due to their integration of hundreds of services and numerous widgets for navigating to other services.

Upon further analysis of the potential consequences of these EPAs, we discovered that some operations could directly pose security threats, e.g., undesirably clicking a payment button, while others might only affect user experience, e.g., mistakenly opening an unwanted page. Figure 7 illustrates a security-related aEPA from `Bilibili` detected by EP-Detector. The functionalities of *one-click for triple hit* (O2T) and *video acceleration* are designed to be triggered by the `longClick` event on two small buttons with overlapped *safe area* (illustrated by the grey dot and red frame in Figure 7(a)). This *unsuitable layout* could result in unintended acceleration (Figure 7(b)) or an irreversible loss of `Bilibili` Coins against one's will (Figure 7(c)).

This inspires us to further explore the types and distribution of the consequences triggered by EPAs. By an in-depth study on the detected EPAs, we conclude 5 categories of potential harmful consequences: *Security (Sec)*, *Functionality (Fun)*, *Advertising (Adv)*, *Efficiency (Eff)* and *User Experience (Use)*.

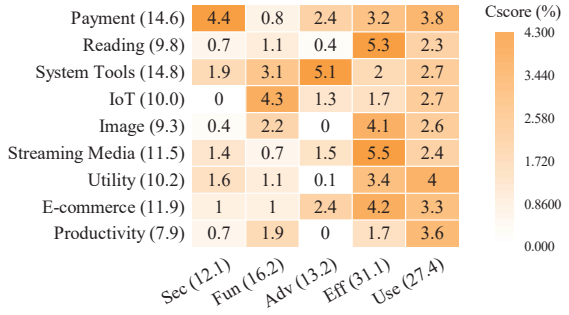


Fig. 8. Relation between Consequences & App Categories.

Combined with the categories of apps, we compute a fine-grained proportion named $CScore_{i,j}$ (defined as Eq.(6)), to indicate the distribution across different types of consequences.

$$CScore_{i,j} = \frac{N_{ij}}{\sum_{i \in CSet, j \in ASet} N_{ij}} \tag{6}$$

where N_{ij} denotes the number of the EPAs in the i-th consequence type and j-th app category; $CSet$ and $ASet$ denote the set of consequence types and app categories respectively. The overall distribution is illustrated in Figure 8. It can be observed that the $Cscore$ of serious consequence categories—Sec 12.1% and Fun 16.2%—are smaller compared to others, but the total counts of these categories (621 and 832) are significant. Although these consequences are only *potentially* triggered, they deserve the attention of both developers and users. We provide related suggestions in the following section. We can also observe that the payment apps (e.g., `Alipay`, `Bankcomm` and `WeChat`) exhibit a higher prevalence with security consequence, accounting for 30.1% among app categories. Interestingly, EP-Detector didn't discover any EPA with advertising consequence in the *Productivity* apps, but detected many such EPAs in *System Tools*. By careful analysis, the advertising consequence tends to be triggered by unintended operations on simplified page with singular functionality. The system tools provide many such scenarios, such as battery testing, file management and network monitoring.

Summary: EPAs are significantly prevalent in current real-world apps (an average of 96.91 per app) with nearly *one EPA for every two widgets*, and their number increases with the complexity of the app pages. The EPAs capable of causing security and functionality consequences are in the minority (28.3%), but their total counts (621 and 832) are considerable.

VII. Threats to Validity

Detection Efficiency. Both widget navigation and simulated operations employed in EP-Detector are time-consuming. Although optimization strategies like page-level abstraction, behaviour grouping for bEPA, and simplified operations for aEPA/eEPA, have been applied, detecting a single page still costs 5.6 minutes on average. Dozens of hours are spent on detecting *large-size* real-world apps, e.g, `WeChat` with 238 pages. Exploring efficient navigation for example using static analysis could scale up the approach as a future improvement.

Failed Pages. Due to delays in page rendering and interruptions from advertising, a small number of pages and their *event traces* cannot be recorded into the *Recorder*, leading to failed detection of these pages. A potential future improvement is to set a *Failed Checker* to detect the page failure and mark the triggering traces for further investigation.

Customized Behaviours. Customized third-party systems and specific apps are not supported in the current version of EP-Detector, due to their rare emergence in practical use. To support them, a way is to add a detection *Dispatcher* compatible to the customized behaviours according to the attributes of the target system and app.

VIII. Related Work

A. GUI Test in Android

The event-driven nature of Android apps increases the complexity of its execution routines, making GUI-based dynamic testing a challenging task. Traditional GUI-based testing approaches [1], [34]–[38] tend to simulate user operations and system events using frameworks provided by Android, e.g., Monkey [39] and its variants [40], [41], to achieve enhanced testing accuracy and optimized performance. To further improve the test coverage and expose more crashing bugs, model-based GUI testing is proposed [19]–[23], [42], [43] to exhaustively explore the the execution paths through abstracting the app behaviours guided by a model and enforcing various user/system interactions. Subsequently, non-crashing functional bugs, stemming from execution logic errors, have received increasing research attention [13]–[15], [33], [44], as they significantly affect user experience [45]. The functional bugs are triggered in rare program paths, which is difficult to capture using the traditional test models. Hence, fine-grained *attribute-targeted* test tactic such as *metamorphic fuzzing* [46]–[48] and *state differential analysis* [14], [15] are employed to generate task-oriented test inputs. The EPAs in this work are anomalies rooted from both crashes and functionality logic errors. The widget-centric navigation is inspired by the existing *attribute-targeted* exploration.

B. Oracle for Android Testing

Dynamic app testing necessitates automated test oracle to explore and confirm the test results efficiently. The test oracle must be adaptable to different testing goals. For instance, to detect security vulnerabilities [7], [16], [17], the oracle often compares the test results with given fingerprint features of the ground-truth in benchmarks. To detect crash bugs [8]–[10], the oracle monitors the sharp changes of the device status and evaluates them with a given threshold. The detection of non-crashing functional bugs [13]–[15], [49] commonly needs to perceive the gap between the page under test and the expected page, requiring the oracle to compare page elements or attributes. For vision-related page checking, the oracle usually employs straightforward visual comparison between screenshots of critical execution points [50]–[55]. These works

maintain independent GUI scripts for the identification of testing targets. The EPA detection is a novel testing problem involving collecting page widgets and monitoring of device environment. Therefore, the test oracle in the EP-Detector integrates multiple comparison strategies targeting at attributes of page widgets and resource status of devices.

IX. Conclusion

This work systematically studies three types of error-prone operation anomalies (EPAs) in Android applications. Using dynamic GUI detection and automatic monitoring, an automated detection tool EP-Detector is developed to facilitate the study of EPAs, which utilizes multiple optimization strategies and EPA-oriented detection designs. Using EP-Detector, 5136 error-prone operations are detected in 53 commonly used real-world Android apps, illustrating the prevalence and severity of the EPAs. As future work, we plan to integrate the static analysis to alleviate the heavy time consumption and support a wider range of operations, such as special operations provided by third-party systems. In addition, we aim to conduct in-depth study on the pages that fail the detection to further enhance the EP-Detector's practicability.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feed back. This work was supported in part by the Natural Science Foundation of Tianjin of China, (Grant No. 23JCYBJC00320, 21JCZDJC00740), Science and Technology Project of Haihe Lab of ITAI, (Grant XCHR-20230701), the National Natural Science Foundation of China (Grant No. 62002177, 62102197) and the Open Fund of Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation (Grant No. CSSAE-2023-001).

References

[1] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011, pp. 77–83.

[2] (2022, Jan.) Airtest project. [Online]. Available: http://airtest.netease.com/

[3] (2022, Jan.) Appium-automation for apps. [Online]. Available: https://appium.io/

[4] (2022, Jan.) Testin-web and mobile application testing. [Online]. Available: http://www.testin.net/

[5] J. Ye, K. Chen, X. Xie, L. Ma, R. Huang, Y. Chen, Y. Xue, and J. Zhao, "An empirical study of gui widget detection for industrial mobile games," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1427–1437.

[6] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 67–77.

[7] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, P. Gill *et al.*, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*, 2018.

[8] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *Proceedings of the IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 2016, pp. 33–44.

[9] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[10] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[11] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 408–419.

[12] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes understanding and benchmarking framework-specific exceptions of android apps," *IEEE Transactions on Software Engineering*, 2020.

[13] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–31, 2021.

[14] J. Wang, Y. Jiang, T. Su, S. Li, C. Xu, J. Lu, and Z. Su, "Detecting non-crashing functional bugs in android apps via deep-state differential analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 434–446.

[15] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su, "An empirical study of functional bugs in android apps," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1319–1331.

[16] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 209–220.

[17] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques." in *NDSS*, 2016.

[18] "Android 10 review," retrived at 21 march 2024. [Online]. Available: https://www.theverge.com/2019/9/4/20848251/android-10-review-dark-theme-focus-mode-gestures

[19] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[20] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[21] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 481–492.

[22] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Guided bug crush: Assist manual gui testing of android apps via hint moves," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–14.

[23] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 469–480.

[24] "View," retrived at 21 march 2024. [Online]. Available: https://developer.android.com/reference/android/view/View

[25] "Widget," retrived at 21 march 2024. [Online]. Available: https://developer.android.com/reference/android/widget/package-summary

[26] "Guesture detector," retrived at 21 march 2024.

[27] "Motion event," retrived at 21 march 2024. [Online]. Available: https://developer.android.com/reference/android/view/MotionEvent?hl=en

[28] "Prototypr.io," retrived at 21 march 2024. [Online]. Available: https://blog.prototypr.io/8-rules-for-perfect-button-design-185d1202ee9c

[29] "Ux planet," retrived at 21 march 2024. [Online]. Available: https://uxplanet.org/7-rules-for-mobile-ui-button-design-e9cf2ea54556

[30] "Make apps more accessible," retrived at 21 march 2024. [Online]. Available: https://developer.android.com/guide/topics/ui/accessibility/apps

[31] J. M. Carroll, "Human-computer interaction: psychology as a science of design," *Annual review of psychology*, vol. 48, no. 1, pp. 61–83, 1997.
[32] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Combodroid: generating high-quality test inputs for android apps via use case combinations," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 469–480.
[33] Google, "Accessibility scanner," Android Developers Official Documentation, 2024, retrieved January 4, 2024, from https://developer.android.google.cn/codelabs/starting-android-accessibility?hl=en0.
[34] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proceedings of the 24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 659–674.
[35] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework." in *NDSS*, 2015.
[36] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: automatic reconstruction of android malware behaviors." in *Ndss*, 2015.
[37] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.
[38] K. Moran, C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk, "Detecting and summarizing gui changes in evolving mobile apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 543–553.
[39] Google. (2022, Jan.) Ui/application exerciser monkey. [Online]. Available: https://developer.android.com/studio/test/monkey.html
[40] ——. (2022, jan) Monkeyrunner. [Online]. Available: https://developer.android.google.cn/studio/test/monkeyrunner
[41] (2022, Jan.) Maxim. [Online]. Available: https://github.com/zhangzhao4444/Maxim
[42] 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*, 2021, pp. 119–130.
[43] J. Yan, S. Zhang, Y. Liu, X. Deng, J. Yan, and J. Zhang, "A comprehensive evaluation of android icc resolution techniques," *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2021.
[44] S. Wang, Y. Wang, X. Zhan, Y. Wang, Y. Liu, X. Luo, and S. C. Cheung, "Aper: Evolution-aware runtime permission misuse detection for android apps," *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 125–137, 2022.
[45] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 738–748.
[46] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, "Understanding and finding system setting-related defects in android apps," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 204–215.
[47] J. Sun, T. Su, K. Liu, C. Peng, Z. Zhang, G. Pu, T. Xie, and Z. Su, "Characterizing and finding system setting-related defects in android apps," *IEEE Transactions on Software Engineering*, 2023.
[48] C. Zhang, Y. Li, H. Chen, X.-F. Luo, M. Li, A.-Q. Nguyen, and Y. Liu, "Biff: Practical binary fuzzing framework for programs of iot and mobile devices," 2021, pp. 1161–1165.
[49] A. S. Alotaibi, P. T. Chiou, and W. G. J. Halfond, "Automated repair of size-based inaccessibility issues in mobile applications," 2021, pp. 730–742.
[50] Y.-D. Lin, E. T.-H. Chu, S.-C. Yu, and Y.-C. Lai, "Improving the accuracy of automated gui testing for embedded systems," *IEEE software*, vol. 31, no. 1, pp. 39–45, 2013.
[51] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, and Y.-C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for android devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, 2014.
[52] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, "Gui-guided test script repair for mobile apps," *IEEE Transactions on Software Engineering*, 2020.
[53] 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*, 2021, pp. 191–203.
[54] R. Coppola, L. Ardito, M. Torchiano, and E. Alégroth, "Translation from layout-based to visual android test scripts: An empirical evaluation," *Journal of Systems and Software*, vol. 171, p. 110845, 2021.
[55] Y. Su, C.-Y. Chen, J. Wang, Z. Liu, D. Wang, S. Li, and Q. Wang, "The metamorphosis: Automatic detection of scaling issues for mobile apps," 2022.