Characterizing and Repairing Color-Related Accessibility Issues in Android Apps

Jiahao Gu
Xiamen University
Xiamen, China
gujiahao@stu.xmu.edu.cn

Huaxun Huang*
Xiamen University
Xiamen, China
huanghuaxun@xmu.edu.cn

***Abstract*—As Android apps become increasingly prevalent in daily life, a common issue in the development process is the configuration of UI colors, leading to color-related accessibility issues that make the text or non-text on the app's UI difficult to see due to low color contrast. Such color-related accessibility issues are among the top issues in apps, having a negative impact on vision and user experience. However, state-of-the-art approaches are based on predefined rules and lack an understanding of strategies for alternative colors, therefore failing to generate patches acceptable to both app users and developers. To address this research gap, we first conducted an empirical study to explore common strategies used by app developers when fixing real-world color-related accessibility issues. Based on these findings, we proposed DroidPalette, an automated approach for repairing color-related accessibility issues in Android apps. DroidPalette encodes the common strategies used by app developers for selecting issue-fixing colors, as identified in our empirical study, and combines this with the candidate issue-fixing attributes identified from the Android framework and third-party libraries to generate patches. We evaluated DroidPalette on 497 color-related accessibility issues across 105 real-world Android apps, achieving a success rate of 66.60%. Encouragingly, out of 13 patches submitted to GitHub repositories, 8 have received positive feedback from app developers.**

***Index Terms*—Android, Automated Repair, Color-Related Accessibility Issues**

I. Introduction

Android apps typically contain a diverse range of user interfaces (UI) to deliver the functionalities designed by app developers [1], [2]. However, despite their widespread adoption, accessibility issues [3]–[7] remain a significant concern in app design and development. Such issues often hinder usability of app users, especially for individuals with disabilities. One of the common issues among them is **color-related accessibility issues**, which occur when the contrast between text or non-text and their background is insufficient for app users to understand the content clearly. The above color-related accessibility issues have been considered by app developers as a challenge because they negatively impact the user experience [8]–[10]. Therefore, app developers should accurately identify and address these issues to enhance usability and accessibility.

Figure 1 shows a UI page from onebusaway-android [11] that suffers from color-related accessibility issues, where the round progress bar has a color-related accessibility issue due to insufficient contrast between the foreground and background colors. This occurs because the app developers did not specify `android:indeterminateTint`, which is a critical attribute for configuring the foreground color of the progress bar. To repair this issue, the app developer explicitly specified the `android:indeterminateTint` attribute in XML configuration files and chose an appropriate value, as shown in Figure 1.

From the perspective of app developers, the manual process of detecting and fixing color-related accessibility issues poses a significant challenge. Android apps typically contain thousands of XML configuration files used to build the app's UI pages, with each XML file containing multiple UI components [12]. Manually inspecting each UI component to identify potential color-related accessibility issues is a time-consuming task. For detecting these issues, there are existing research efforts to assist [4], [13]. Among them, Xbot is the state-of-the-art approach that relies on Google Accessibility Scanner [13] to detect color-related accessibility issues. Specifically, Xbot performs a static analysis on all XML configurations in the app and calculates the contrast ratio between foreground and background colors to identify color-related accessibility issues (Output example is shown in Figure 2). However, these approaches cannot provide developers with patches for these color-related accessibility issues. Given the prevalence of such issues, repairing these identified issues still requires a significant manual effort, especially when there are potentially numerous color-related accessibility issues in an Android app. Therefore, an automated approach is needed to help developers automatically repair these color-related accessibility issues.

However, to automate the repair of color-related accessibility issues, one should address the following research challenges. First, how to **choose colors** that eliminate low color contrast and align with the styles of other UI components. Second, there are **multiple ways to fix color-related accessibility issues**. Given the large codebase of Android apps, finding the right place to implement the fix in the app code can be difficult. Additionally, ensuring that the patch does not cause any side effects is also a crucial point. To further repair the color-related accessibility issue shown in Figure 1, one first needs to comprehensively consider and select `@color/header_text_faded_color` as the issue-fixing color. Then, we need to analyze the

*Huaxun Huang is the corresponding author of this paper.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00169

(a) (b)

route_info_head.xml

```
01 <ProgressBar
02       android:id="@+id/route_info_loading_spinner"
03       style="?android:progressBarStyleLarge"
         ...
04       android:visibility="visible"
05 +     android:indeterminateTint="@color/header_text_faded_color"
06       android:layout_alignParentLeft="true"
07       android:layout_centerVertical="true"/>
```

(c)

Fig. 1: This issue originates from the onebusaway-android [11]. Specifically: (a) A UI component with color-related accessibility issue, (b) The developer's fix, (c) Edit `route_info_head.xml` to repair the color-related accessibility issue in (a).

Android project's codebase to locate the effective attribute `android:indeterminateTint` in order to determine where our patch should be applied.

Existing studies [14]–[16] have extensively explored color-related accessibility issues from multiple perspectives. For example, Linares-Vásquez et al. [14] developed a method for generating a color scheme aiming to reduce energy consumption in app UIs rather than address color-related accessibility issues. Recently, Zhang et al. [15] proposed Iris, an automated approach to repair color-related accessibility issues. However, Iris generates repair patches for these issues using hard-coded rules, which limits its ability to comprehensively cover all possible scenarios. Additionally, Iris's repair process relies on a corpus of UI colors built from other Android apps. Directly reusing the color schemes from different apps does not fully consider the style of the page where the UI widget to be repaired is located, resulting in many patches generated by Iris still failing to meet contrast requirements. So far, no existing research efforts have been proposed to understand the common practices of app developers in repairing color-related accessibility issues.

To address the research gap mentioned above, we conducted an empirical study on 202 real-world color-related accessibility issues. Our aim was to understand the strategies that app developers use to fix these issues, focusing on how they modify the app code and select appropriate colors for issue repair. Based on the above findings, we propose DroidPalette, an automated approach for repairing color-related accessibility issues in Android apps. Specifically, DroidPalette selects issue-fixing colors for buggy UI components by referencing UI components with similar attribute configurations on a UI page. This is mainly because 91.09% (=184/202) of the issues in our empirical dataset were resolved using this strategy. To generate effective patches that can adjust the color of buggy UI components, DroidPalette utilizes knowledge from the Android framework and third-party libraries to create a ranked list of attributes that can be potentially used to fix the issue. The generated patches are further validated by applying them and observing their runtime visual appearance.

We applied DroidPalette on 105 real-world Android apps to evaluate its effectiveness. The experimental results showed that DroidPalette can successfully repair 331 out of 497 valid issue reports generated by Xbot, achieving a success rate of 66.60% in issue repair. To investigate whether our patch is acceptable to app developers and users, we conducted a user survey on open-source Android apps to collect feedback from both developers and users. We found that DroidPalette received positive feedback from app developers and users on fixing color-related accessibility issues, indicating that DroidPalette can efficiently resolve such issues. So far, we have submitted 13 pull requests, among which 8 have already confirmed our fixes, showing the usefulness of DroidPalette.

In summary, our main contributions are as follows:

- To the best of our knowledge, we are the first to conduct an empirical study on the repair of color-related accessibility issues.
- We propose DroidPalette, an approach that can eliminate color-related accessibility issues in Android apps.
- We evaluated DroidPalette on real-world Android apps, and the results demonstrate its effectiveness and usefulness in repairing color-related accessibility issues.

II. Background

A. Android UI

The user interface (UI) of an Android app is composed of View and ViewGroup components structured in a tree-like hierarchy. Typically, an app contains one or more Activities (screens), each of which consists of multiple instances of View and ViewGroup. Specifically, ViewGroup serves as the container for other views and to define layout attributes that determine the positioning of child views on the screen. On the other hand, View is the core UI element used to define UI components such as TextView, ImageView, and others. App developers usually control the runtime behavior, such as the visual appearance, of View and ViewGroup by modifying the attribute configurations of XML elements within project XML files.

B. Color-Related Accessibility Issues

In Android app design, color plays a crucial role not only in aesthetics but also in ensuring accessibility. Poor color contrast can significantly hinder the user experience for elderly users, people with visual impairments, or those with other vision-related disabilities. To meet accessibility requirements, the Web Content Accessibility Guidelines (WCAG) [17] have specified that the contrast ratio for **regular text** (text smaller than 18pt or bold text smaller than 14pt) should be at least 4.5:1. For **large text** (text larger than 18pt or bold text larger than 14pt) or **non-text elements** (e.g., images), the minimum contrast requirement is 3:1. However, when the contrast of the UI components defined within a UI page does not meet the requirements, users may be unable to identify key information

```
Image contrast
OneBusAway:id/route_info_loading_spinner
The item's text contrast ratio is 1.07. This ratio is based on an estimated
foreground color of #5DB53B  and an estimated background color of #4CB050.
Consider increasing this item's text contrast ratio to 3.00 or greater.
```

Fig. 2: The issue report about Figure 1(a) generated by Xbot.

in the image, causing color-related accessibility issues that pose a major barrier for app users.

III. Motivating Example

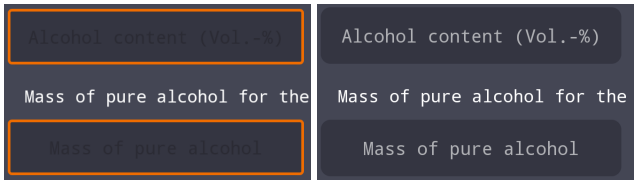
In this section, we focus on evaluating the state-of-the-art methods to identify their limitations in repairing color-related accessibility issues. We chose Iris [15], which is the state-of-the-art approach to repair color-related accessibility issues in Android apps, as the baseline. Specifically, the issue-repair process of Iris relies on a database that contains color schemes collected from a large corpus of open-source and closed-source Android apps. It then uses these schemes along with a set of hardcoded rules to edit the app's XML configuration files and generate patches. Additionally, we aim to explore the capability of large language models in repairing color-related accessibility issues. We selected GPT-4o [18], a popular multimodal large language model that can process both text and images as inputs to complete specific tasks. Specifically, we provide an issue report that includes the view hierarchy of the UI pages to pinpoint the buggy UI components, screenshots of the UI pages, and the corresponding XML configuration files. GPT-4o then generates patches for the XML configuration files to repair the color-related accessibility issues.

We applied Iris and GPT-4o to the issue illustrated in Figure 1 in order to evaluate their effectiveness based on two criteria: their ability to select appropriate colors for issue repair and their effectiveness in generating plausible patches to adjust the colors of buggy UI components. This allows us to understand the limitations of existing approaches and better illustrate the intuition of our proposed approach.

A. Color Selection Process

For the issue shown in Figure 1, Iris chose black to fix the buggy progress bar. The result, shown in Figure 4(b), differs from the solution provided by the app developers. This is because Iris's issue repair process mainly relies on selecting colors from a reference database. Although Iris considers the harmony of the UI page when selecting an issue-fixing color, if the developer's desired color is not in the database, Iris cannot find that color to use as the issue-fixing color. Our user study in Section VI also highlights this issue.

For GPT-4o, despite providing contextual information such as the view hierarchy and screenshots of the problematic UI components and the UI page, we found that the colors selected by GPT-4o to fix color-related accessibility issues still have the following limitations: (1) Due to the randomness of large language models, GPT-4o does not consistently produce the same results; (2) Although it can consistently produce the same results for some issues, the lack of understanding of app developers' strategies for fixing color-related accessibility



(a) (b)

activity_main.xml

```
01 <EditText
02       android:id="@id/editTextProzent"
03       android:layout_height="48dp"
04       android:layout_width="60dp"
// An incorrect patch generated by GPT-4o
05 +     android:textColor="#E6E6E6"
// A correct patch generated by DroidPalette
06 +     android:textColorHint="#E6E6E6"
         ...
07       android:hint="@string/percent_alcohol_edittext"/>
```

(c)

Fig. 3: This issue originates from the VPC [19]. Specifically: (a) UI with color-related accessibility issues, (b) The UI after being repaired with the correct patch, (c) Different patches generated by GPT-4o and DroidPalette.

issues leads to solutions that differ from those provided by the developers, and sometimes even fail to meet the contrast standards specified by WCAG. For example, the color chosen by GPT-4o, as shown in Figure 4, failed to resolve the accessibility issues caused by color contrast.

To address the aforementioned limitations, DroidPalette analyzes real-world patches for color-related accessibility issues, distills the actual color adjustment preferences adopted by developers, and incorporates these insights into its color selection strategy. Specifically, when selecting issue-fixing colors, DroidPalette not only considers the original color scheme of the buggy UI component, but also examines the color usage of other components on the same page. This ensures that the chosen colors not only meet contrast requirements but also better maintain consistency with the overall color scheme of the UI page. For instance, when resolving the issue demonstrated in Figure 1(a), DroidPalette comprehensively evaluates both the original color scheme and color harmony with other page components, ultimately adopting the solution consistent with the developer's fix shown in Figure 1(b) (see Section IV-B for details).

B. Patch Generation Process

We further examined the capabilities of Iris and GPT-4o in generating patches that can adjust the color of buggy UI components at the code level. Specifically, Iris integrates only a set of predefined rules to generate a patch. Consequently, Iris fails to adjust the color of the progress bar because the predefined rules do not cover all possible attributes (such as `android:indeterminateTint`) that can lead to color-related accessibility issues. As for GPT-4o, although it can recommend `android:indeterminateTint` as the issue-fixing attribute, it still suffers from hallucinations and generates incorrect patches. Figure 3 shows such a case from VPC [19], where GPT-4o leverages `android:textColor`

Fig. 4: (a) The effectiveness of GPT-4o's color selection for the issue shown in Figure 1. (b) The effectiveness of Iris's color selection for the issue shown in Figure 1.

instead of `android:textColorHint` to adjust the insufficient color contrast between text hints and the background color.

To address the aforementioned issue, DroidPalette first recommends candidate issue-fixing attributes from the Android framework and third-party libraries that can potentially be leveraged to fix color-related accessibility issues. Then, in the process of generating patches, DroidPalette dynamically monitors the visual appearance of buggy UI components to determine whether the selected candidate attributes can be used to adjust the color of the buggy UI components. Compared to Iris, DroidPalette eliminates the need for manually defining rules on the issue repair process. For example, when fixing the issue shown in Figure 3, DroidPalette successfully identified the `android:textColorHint` attribute from the Android framework as effective to change the color of the buggy text contrast.

IV. Color Selection Strategy of App Developers

To the best of our knowledge, there is no existing effort exploring developers' preferences when choosing colors to fix color-related accessibility issues. Thus, we conducted an empirical study to understand the strategies developers use to address these issues.

A. Dataset Collection

We selected open-source Android projects from GitHub as our study subjects because these projects make publicly available the issue reports and their patches. Our data collection process is as follows: First, we searched the entire GitHub platform using the keyword "`color fix`" to identify pull requests merged by developers. We restricted the programming languages of the pull requests to Kotlin, Java, and XML to ensure their relevance to Android apps. We excluded projects with fewer than 100 stars, resulting in a total of 1,629 commits being collected. Next, we manually excluded pull requests that were unrelated to color-related accessibility issues in Android projects. In the end, we identified a total of 202 pull requests across 63 projects. The selected projects demonstrate diversity in terms of popularity and scale, with the average number of stars for the projects being 3,171 (ranging from 110 to 50.4K), and the average code size being 244K LOC (ranging from 6.9K to 1.9M LOC).

We conducted the data analysis on the aforementioned 202 color-related accessibility issues as follows. Initially, we randomly selected a sample of 101 issues, representing 50% of the total dataset. Two authors, each with two years of experience in Android app development, independently examined the code revisions and associated issue reports to pinpoint code snippets pertinent to the patches. A preliminary taxonomy was developed by compiling the findings of both authors and resolving discrepancies through discussions. Subsequently, the authors labeled the remaining 101 issues in an iterative manner, engaging in discussions to refine the preliminary taxonomy and address any conflicts. The final results were achieved once both authors reached an agreement on the taxonomy and the labeling of the empirical dataset.

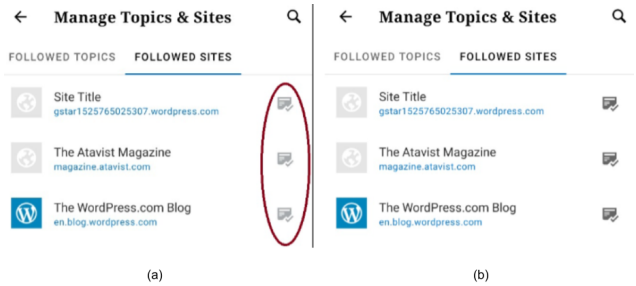


Fig. 5: The issue originates from the WordPress [20]. Specifically: (a) UI page with color-related accessibility issues, (b) The issue repair by adjusting the lightness.

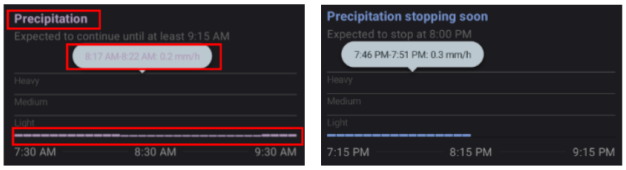
B. Results

We analyzed these 202 identified patches and summarized the developers' choice of issue-fixing colors, which are categorized as T1 to T3.

T1: Selecting Colors within the UI Page. We identified 150 issues where the issue-fixing color was chosen within the same page. This strategy ensures that the selected issue-fixing color can maintain visual consistency with the original UI page. As illustrated in Figure 1, to fix the color of the buggy progress bar, app developers chose white, referencing the color of other UI components with similar visual features on the same UI page, as the issue-fixing color.

Based on the location of the UI component where the issue-fixing color is applied, we further derive the following two issue-fixing color selection strategies of app developers:

- **T1.1: UI Component with Same Alignment.** 129 issues were resolved by selecting colors from UI components that were aligned with the problematic ones. Typically, UI components that are aligned together either share the same parent XML element or are located close to each other in the view hierarchy. This approach works because UI components that meet these criteria usually have similar visual feature definitions, which help maintain a consistent color appearance with other UI components on the same page.
- **T1.2: UI Component with Similar Attribute Configuration.** 104 issues were addressed by selecting colors from UI components with similar attribute configurations in XML files. Components meeting these criteria generally share



(a) (b)

Fig. 6: This issue originates from the breezy-weather [21]. Specifically: (a) UI with color-related accessibility issues, (b) Repair color-related accessibility issues by selecting colors outside the UI page.

similar visual feature definitions, which help preserve a consistent color appearance.

It should be noted that the two aforementioned strategies can be implemented simultaneously by developers, so the sum of these two subcategories does not necessarily match the number of issues in T1.

T2: Adjusting the Lightness of the Issue-Inducing Color. We identified 34 issue reports where app developers repaired color-related accessibility issues by adjusting the lightness of the issue-inducing color. The chosen issue-fixing color stays within the same hue as the issue-inducing color, thus preventing any disharmony with the UI page's overall color scheme. For example, the issue in Figure 5, extracted from WordPress [20], shows such a case where the app developer changed the icon's color from light gray to dark gray in (a), thereby enhancing the icon's contrast against the white background.

T3: Introducing New Colors. We identified 18 issue reports where app developers introduced issue-fixing colors that were not present in the UI page. Figure 6 shows such an example originates from breezy-weather [21], where the app developers used light blue instead of light purple to enhance the contrast of the UI component against a black background. We categorize this as T3, because this light blue does not appear in the UI page.

V. Approach

In this paper, we propose DroidPalette, a dynamic approach for repairing color-related accessibility issues in Android apps. DroidPalette works by editing attribute configurations in XML files, which are primarily used to construct the apps' UI. We chose to adopt such a strategy because 71.29% (144/202) of the empirical dataset utilized this approach, in line with the practices established by Zhang et al. [15].

Prior work [15] on repairing color-related accessibility issues has the following major limitations: (1) their color selection strategy relies on a database of UI components' colors collected from a large corpus of Android apps, which cannot provide the fixes aligned with app developers' expectation when the issue-fixing color is not present in this database, and (2) the patch generation process relies on hardcoded rules, therefore lacking a comprehensive understanding of all possible attributes that could be used to fix color-related

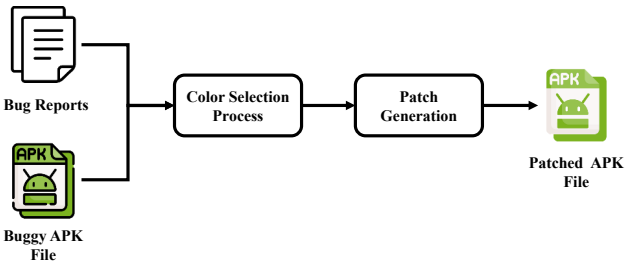


Fig. 7: Overview of the DroidPalette.

Algorithm 1: Color Selection Process

Input: View hierarchy VH of the page P, buggy UI component u, Color Reference Database DB
Output: The issue-fixing color c_{best}

$\mathcal{C} \leftarrow \emptyset$;
foreach u_p *located in* P **do**
  Add the foreground color of u_p to $\mathcal{C}$;
$C \leftarrow C \cup \text{TopN}_{c \in DB}\, Score(u, c)$;
$c_{best} \leftarrow$ null;
$score = 0$;
foreach $c \in \mathcal{C}$ **do**
  Adjust the lightness of c;
  if $score < Score(u, c)$ **then**
    $c_{best} \leftarrow c$;
    $score = Score(u, c)$;
return c_{best};

accessibility issues, making it impossible to change the color of buggy UI components by modifying the XML configuration files. In light of the above limitations, DroidPalette takes the following actions: (1) encoding the common color selection strategies observed in Section IV to generate issue-fixing colors, and (2) identifying candidate issue-fixing attributes from the Android framework and third-party libraries, finalizing the generated patch based on the runtime behavior of these candidate attributes.

The overview of DroidPalette is as follows. Given the app under test (AUT) and its corresponding issue reports generated by Xbot, which includes the UI screenshots, the identified foreground and background colors of buggy UI components, and the view hierarchy of UI pages where the buggy UI component locates, as inputs, DroidPalette outputs a patched APK file that eliminates the color-related accessibility issue in the buggy UI components. The issue-fixing process in DroidPalette consists of two main stages. In the first stage, DroidPalette selects and fine-tunes a ranked list of candidate colors to choose the optimal issue-fixing colors. In the second stage, DroidPalette selects candidate issue-fixing attributes from the Android framework and third-party libraries, and monitors the visual appearance of each generated patched XML file to fix color-related accessibility issues.

A. Color Selection Strategy

DroidPalette first selects an issue-fixing color for UI components with color-related accessibility issues. As discussed in Section IV, the issue-fixing color is typically either derived from the color of the same UI page or achieved by adjusting the lightness of that color. This approach ensures that the selected issue-fixing color aligns with the style of the UI pages. Therefore, DroidPalette incorporates the common strategy used by app developers for choosing candidate issue-fixing colors, using a fitness score to evaluate how well the chosen issue-fixing color blends with the overall style of the UI page. In case when DroidPalette fails to identify issue-fixing color from the above process, DroidPalette integrate Iris's reference database as an extra list of candidate issue-fixing color to address the issues related to T3.

Algorithm 1 shows the process of DroidPalette in selecting issue-fixing colors. There are two steps involved: (1) DroidPalette first generates a set of candidate issue-fixing colors that not only can improve the color contrast of UI pages but also align with the original design style of the original UI components (Lines 2-4); (2) DroidPalette then keeps adjusting the lightness of each candidate issue-fixing color, and encodes a fitness score to evaluate overall consistency and harmony of the page design, and to output an issue-fixing color with the highest fitness score (Lines 7-11).

Fitness Score Calculation. To evaluate whether a candidate color can match with other colors on a UI page, a fitness score is proposed. The design of the fitness score is based on our empirical finding that *app developers typically consider issue-fixing colors by considering the alignment, color, and attribute configuration similarity of UI components in the same page*. Specifically, for a given issue-fixing color c of a buggy UI component u located in the UI page P, the fitness score $Score$ is calculated by the average of similarities between u and all the UI components u_P in U_P (all UI components in P) as follows:

$$Score(c,u) = \frac{1}{|U_P|} \sum_{u_P \in U_P} \big(P(u,u_P) \times A(u,u_P) \times C(c,u_P)\big) \tag{1}$$

Specifically, P measures the distance between u and u_P in the view hierarchy of the page, based on the intuition that UI components positioned closer together are more likely to share similar colors. Therefore, P is calculated by measuring the edit distance $Dist$ of view hierarchy VH in P as follows:

$$P(u_1,u_2) = \frac{1}{1 + Dist(u_1,u_2,VH)} \tag{2}$$

$A(u_1,u_2)$ measures the similarity in attribute configuration by calculating the proportion of shared attributes between u_1 and u_2 as follows:

$$A(u_1,u_2) = \frac{|A(u_1) \cap A(u_2)|}{|A(u_1) \cup A(u_2)|} \tag{3}$$

$C(c,u)$ calculates the color similarity between c and the foreground color c_f of u as follows:

$$C(c,u) = \frac{Contrast(c,u)}{1 + D_{HSL}(c,u.c_f)} \tag{4}$$

Here, $Contrast(c,u)$ denotes the color contrast of u when applying c as the foreground color. $D_{HSL}(c_1,c_2)$ represents the differences in hue, saturation, and lightness, reflecting human color perception as per the practices of Zhang et al [15].

Candidate Issue-Fixing Color Selection. With the above fitness score, DroidPalette then selects appropriate colors for repairing color-related accessibility issues in the UI based on the feedback of fitness score. DroidPalette first chooses colors from the *page* and *reference database* of Iris [15] as candidate colors, and then calculates the scores of these candidate colors using Equation 1. When using the reference database, DroidPalette selects the top-N (N is set to 5 by default) colors according to the fitness score illustrated in Equation 1 and adds them to the candidate color set C (Line 4).

DroidPalette then adjusts the lightness of the candidate color for fine-tuning, following our research findings in Section IV. Specifically, DroidPalette adds a random delta value in a small value range (i.e., $(0, d\%]$) to its lightness value from the starting point. Initially, d is set to 1. DroidPalette expands d exponentially if the *Score* is improved. If the *Score* is not improved, DroidPalette switches to the opposite direction (i.e., $[-d\%, 0)$ value range) from the current value and resets d to 1. A candidate issue-fixing color is generated if there is no fitness score improvement in all exploratory directions (i.e., local optima). When the lightness adjustment reaches a critical threshold ($\leq 10\%$ or $\geq 90\%$), the original color characteristics can be lost, and the color is therefore considered invalid. Finally, among the colors that satisfy the color contrast criteria of WCAG, DroidPalette selects the issue-fixing colors that achieve the highest fitness score.

B. Patch Generation

DroidPalette then aims to apply the selected issue-fixing color in the app's XML configuration files to generate a patch. Algorithm 2 illustrates the process. Specifically, given the buggy UI component U in the view hierarchy VH as inputs, DroidPalette identifies a set of candidate XML elements X associated with U (Line 1). Then, for each XML element $x \in X$, DroidPalette attempts to identify a set of candidate issue-fixing attributes $A = \{a_1, \ldots, a_n\}$ from both the Android framework and the third-party libraries (Lines 3-4). At last, DroidPalette evaluates the visual effects after applying the issue-fixing color to each candidate issue-fixing attributes in order to generate the patch (Lines 5-9).

For the UI component U, DroidPalette first identifies a set of XML elements $X = \{x_1, ..., x_n\}$ that can affect the runtime behavior of U. Specifically, X includes the following types of XML elements.

- The XML element X_u that defines the UI component U. Specifically, X_u can be localized by matching (1) IDs, and (2) text between view hierarchy and the XML configuration

Algorithm 2: Patch Generation Process

Input: View hierarchy VH, target UI component U
Output: Generated patch $Patch$

```
X ← findCandidateXMLElements(VH, U);
foreach x ∈ X do
    A ← identifyFixAttributes(x);
    foreach a ∈ A do
        c_new ← computeIssueFixingColor(x, a);
        x' ← applyColorChange(x, a, c_new);
        if evaluateVisualEffect(VH, x') then
            P ← generatePatch(x, a, c_new);
            return Patch;
return ∅ ;
```

files, following the existing practices proposed by Zhang et al. [15].

- The parent nodes of X_u. This is primarily because the configuration of attributes in parent nodes directly affects the performance and behavior of child nodes.
- The XML elements that are referenced by X_u. For example, an XML element can specify `android:src="@drawable/d"` to reference the drawable element with the ID named `d`, which is defined from other XML configuration file.

After successfully locating the candidate XML elements, DroidPalette further identify candidate issue-fixing attributes A for each XML element x. An attribute a will be included in A if (1) it is configurable in x, and (2) if it accepts values in the color data format. Such the above design is based on the issue-fixing practices of 71.29% (144/202) issues we observed in the empirical dataset. To achieve this goal, DroidPalette first collects a list of all candidate attributes that meet the above conditions from the Android framework and third-party libraries on which the app depends. Sepcifically, the candidate issue-fixing attributes are obtained using the method proposed by ConfDroid [12], which performs static slicing on the code that processes XML elements within the Android framework and third-party libraries to identify candidate list of color-related attributes related to the color of the issue-inducing UI component. Therefore, compared with existing approach [15], DroidPalette does not require manually maintaining these rules in case when the Android framework or its third-party libraries are evolved. Then, DroidPalette assigns the value of the selected issue-fixing color to these candidate attributes one by one and dynamically tests whether the candidate attribute can successfully modify the color of the buggy UI component.

It is noted that DroidPalette only attempts to use one issue-fixing attribute in the generated patches. It shows threats that, in practice, there are issues whose fix relies on multiple issue-fixing attributes. However, we did not find such cases in our empirical dataset.

VI. Evaluation

We implemented DroidPalette based on UIAutomator2 [22], an open-source UI testing framework for Android apps. We explore the following research questions to evaluate DroidPalette:

- **RQ1 (Effectiveness):** What is the effectiveness of DroidPalette and the baseline methods in fixing color-related accessibility issues in Android apps?
- **RQ2 (Ablation Study):** How do DroidPalette and the baseline methods perform specifically in the color selection and patch generation processes?
- **RQ3 (Usefulness):** Are the patches generated by DroidPalette well-received by both Android app developers and users?

A. Evaluation Subjects

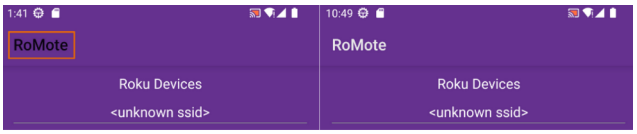
We collected the evaluation subjects as follows. First, we selected 650 apps from the F-Droid to serve as our initial dataset. To further refine the dataset, we only kept apps with more than 10 stars to ensure the popularity of our selected app subjects. The above process in total results in a final dataset of 105 apps. The distribution of stars of these selected apps is wide-ranging, with an average of 857. Additionally, 30 apps have 500+ stars, and 17 apps have 1,000+ stars.

We ran Xbot on the collected 105 apps, resulting in a total of 497 issue reports. These reports were then categorized according to the color contrast requirements for different types of UI components as specified in WCAG [17]. This resulted in 340 issue reports for regular text, 73 issue reports for large text, and 84 issue reports for non-text elements (i.e., meaningful images). Then, we ran Xbot on the repaired APK files. If Xbot no longer reported an issue, the issue was considered fixed. Such the above process is based on existing practices for handling UI accessibility issues [15]. We avoided using manual annotation due to its subjectivity.

B. Baselines

We compared DroidPalette with the following baselines:

- **Iris:** The state-of-the-art approach for repairing color-related accessibility issues [15]. The issue repair process of Iris primarily relies on a referenced database extracted from a large number of apps.
- **GPT-4o:** One of the most popular multi-modal large models released by OpenAI [18]. We evaluated the effectiveness of GPT-4o by providing it with (1) an issue report, (2) the buggy XML configuration file, and (3) a screenshot, and then allowing GPT-4o to generate a patch for the buggy XML configuration files. We repeat GPT-4o three times to reduce the randomness of its output, and consider GPT-4o successfully repairs an issue if it consistently output the same result that eliminate the color-related accessibility issues. The prompt is set as follows:



(a) (b)

Fig. 8: This issue originates from the RoMote [23]. Specifically: (a) UI with color-related accessibility issue, (b) Repair this color-related accessibility issue by DroidPalette.

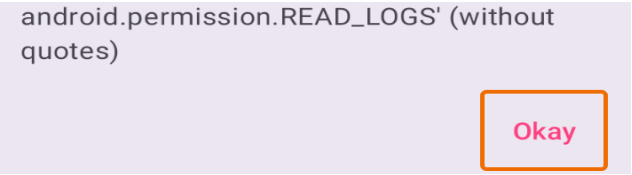
> *The highlighted UI components in the image have color-related accessibility issues. The detection report provides details on these components, and the XML files include their layout and resource information. Please propose a patch on the XML file with replacement colors ensuring at least a 3:1 contrast ratio for large text and non-text elements, and at least 4.5:1 for regular text.*

C. RQ1: Effectiveness

Effectiveness of DroidPalette. Table I shows the results of the study. DroidPalette successfully repaired 331 out of 497 identified color-related accessibility issues, achieving a success rate of 66.60%. Specifically, it fixed 245 issues of regular texts, 51 issues of large texts, and 35 issues of non-texts. Figure 8 provides an example originates from RoMote [23]: in Figure 8(a), the original issue was a black title on a purple background, which made the text difficult to read. DroidPalette addressed this by changing the issue-inducing color to white, along with other text on the same UI page. These results indicate that DroidPalette is effective at improving insufficient color contrast in the UI components of Android apps.

On the other hand, DroidPalette failed to repair the remaining 166 issues, including 95 issues of regular texts, 22 issues of large texts, and 49 issues of non-text elements. Among these, there are issues that could not be fixed because the Android framework or third-party libraries does not provide a publicly-available XML attribute to adjust the color. Figure 9 from EnforceDoze [24] illustrates such an example. As shown in Lines 10-12, app developers fixed the issue by adjusting the app' code of creating and customizing alert dialogs using Material Design styles, where the "Okay" button of the dialog is controlled by the Android framework. Therefore, DroidPalette fails to fix this issue since the confirm button is a pre-defined component of the Android framework and does not provide an XML attribute to adjust its color. The above example also highlights the challenges of fixing bugs at the Java/Kotlin source code level: (1) accurately locating the issue, and (2) handling complex API invocations. Given that this repair strategy is relatively complex and is only required in a small fraction of real-world development scenarios, we leave the exploration of this approach to future work.

Effectiveness of Baselines. Table I shows the effectiveness of Iris and GPT-4o in repairing color-related accessibility issues. Specifically, Iris successfully repaired 198 color-related accessibility issues, achieving a success rate of 39.84%. Iris



(a)

```
01  MaterialAlertDialogBuilder builder = new MaterialAlertDialogBuilder(LogActivity.this);
02  builder.setTitle(getString(R.string.error_text));
03  builder.setMessage(getString(R.string.read_logcat_su_not_avail_or_denied_error_text));
04  builder.setPositiveButton(getString(R.string.okay_button_text), new
        DialogInterface.OnClickListener() {
05      @Override
06      public void onClick(DialogInterface dialogInterface, int i) {
07              dialogInterface.dismiss();
08      }
09  });
10 + androidx.appcompat.app.AlertDialog dialog = builder.show();
11 + Button positiveButton = dialog.getButton(AlertDialog.BUTTON_POSITIVE);
12 + positiveButton.setTextColor(Color.parseColor("#000000"));
```

(b)

Fig. 9: This issue originates from the EnforceDoze [24]. Specifically: (a) The "Okay" button in the alert dialog, (b) The Material Design style implementation that controls the "Okay" button.

Fig. 10: This issue originates from the ttrss-reader-fork [25]. GPT-4o recommended white as the color, which had no effect on the buggy UI component.

failed to repair 133 issues that were successfully repaired by DroidPalette due to the following two reasons: (1) Iris's hardcoded rules for generating patches are only designed for five basic components (e.g., `TextView`, `Button`, etc.). For UI components from the Android framework components and third-party libraries, Iris cannot correctly identify the valid issue-fixing attributes, resulting in failures for repairing color-related accessibility issues; (2) the effectiveness of Iris's repairs depends on the completeness of its reference database. When the reference database lacks suitable alternatives, it leads to the failure of the issue-fixing color selection strategy (see Section III for more details), and may even cause the issue-fixing color found in the database to not meet the color contrast requirements of WCAG.

As for GPT-4o, it can successfully repair 129 color-related accessibility issues, achieving a success rate of 25.96%. On the other hand, GPT-4o failed to generate patches for 368 color-related accessibility issues due to the following reasons: First, despite receiving the same results for multiple issues, the generated issue-fixing colors still did not eliminate the issues caused by insufficient color contrast. Figure 10 shows an example of such cases. Specifically, in the UI page from ttrss-reader-fork [25], the "CLOSE" button and the "WANT TO DONATE?" button has color-related accessibility issues. However, the issue-fixing color recommended by GPT-4o is the same as the original color (both are white) in three runs. Second, due to the randomness of GPT-4o's output, even though we explicitly provided screenshots, UI hierarchy, and

TABLE I: The repair results of DroidPalette, Iris and GPT-4o

Type	Dataset	DroidPalette	Iris	GPT-4o	DroidPalette$_c$	Iris$_c$	GPT-4o$_c$	DroidPalette$_p$	Iris$_p$	GPT-4o$_p$
Total	**497**	**331**	**198**	**129**	**497**	**457**	**344**	**331**	**230**	**192**
Regular Text	340	245	180	122	340	318	278	245	199	167
Large Text	73	51	18	5	73	60	33	51	30	22
Non-Text	84	35	0	2	84	79	33	35	1	3

asked GPT-4o to repair color-related accessibility issues in a buggy UI component, GPT-4o fails to consistently output the results that can eliminate the insufficient color contrasts when repeating the experiments three times. Third, when generating patches, GPT-4o tends to hallucinate by using attributes that cannot actually adjust the visual appearance of buggy UI components. The screenshot and the code changes illustrated in Figure 3 shows a case extracted from VPC [19] where GPT-4o fails to patch. As we can see, there are two color-related accessibility issues induced by the hint colors in edit texts, which have insufficient color contrast. However, GPT-4o chose `android:textColor`, which cannot actually adjust the text hint color in edit texts. On the other hand, DroidPalette can successfully repair this issue by choosing the attribute `android:textColorHint` to adjust the above hint colors.

The above results show that DroidPalette outperforms the baselines in terms of the number of color-related accessibility issues that were successfully repaired. As discussed in Section III, for each color-related accessibility issue report, DroidPalette first identifies UI components with similar visual features on the UI page and recommends an issue-fixing color. Then, it generates patches by using attributes from the Android framework and third-party libraries. By doing so, DroidPalette can consistently generate patches that align the style of the UI page while eliminating insufficient color contrast.

D. RQ2: Ablation Study

DroidPalette consists of two main stages, so we designed an ablation experiment combining baselines to evaluate the results of these two stages separately. Furthermore, in the ablation study for the Color Selection Strategy, we specifically analyzed the usage ratios of different color selection strategies (T1-T3) in detail.

1) Color Selection Strategy: We first evaluate the effectiveness of the color selection strategy between DroidPalette and baselines. Specifically, for each issue in the evaluation dataset, we configured DroidPalette and the baselines to generate the issue-fixing color only. We then manually adjusted the XML configuration of the buggy UI components to check whether the generated issue-fixing color could eliminate the insufficient color contrast. We denote the results as DroidPalette$_c$, Iris$_c$, and GPT-4o$_c$. It is worth noting that here we only focus on whether the selected issue-fixing color can improve the color contrast of the buggy UI component. We provide a detailed comparison of the user preferences for selected issue-fixing colors among different approaches in RQ3.

The results are shown in the Table I. DroidPalette combines the results of our empirical study with Iris's proposed reference database scheme, achieving a 100% success rate on the experimental dataset. Specifically, 327 color-related accessibility issues were resolved using T1, 153 were resolved using T2, and the remaining 17 issues were resolved by T3. Moreover, Iris provided 457 valid replacement colors out of 497 color-related accessibility issues, achieving a success rate of 91.95%. In contrast, GPT-4o provided 344 valid replacement colors, with a success rate of 69.22%. The above results shows the effectiveness of DroidPalette in choosing valid issue-fixing colors comparing to the baselines.

2) Patch Generation: We then proceed to evaluate the effectiveness of the patch generation process between DroidPalette and baselines. Specifically, given the XML element of the buggy UI, we let each approach recommend issue-fixing attributes, then filled in randomly generated colors to check whether the recommended issue-fixing attributes could be applied to change the color of the buggy UI component and eliminate insufficient color contrast. We denote the results as DroidPalette$_p$, Iris$_p$, and GPT-4o$_p$.

The results are shown in Table I. As we can see, DroidPalette achieved higher success rates than Iris and GPT-4o across different types of UI components. Overall, DroidPalette successfully identified 331 valid attributes in the Patch Generation stage, achieving a success rate of 66.60%. In comparison, Iris and GPT-4o successfully identified 230 and 192 valid attributes, with success rates of 46.28% and 38.63%, respectively. The above results show the effectiveness of DroidPalette in generating correct patches comparing to the baselines.

E. RQ3: Usefulness

To address RQ3, we examine the usefulness of patches generated from both DroidPalette and the baselines from two dimensions: app users and app developers.

User Study. We conducted a user-based study where we asked users to compare the original and repaired UIs of DroidPalette and the baselines. The goal of this user study was to understand how our repairs can be acceptable from a user's perspective to eliminate color-related accessibility issues. This survey covers repair patches for 10 typical color-related accessibility issues, with each corresponding UI page containing one or more issues. The issues in the survey were selected as follows. First, we filtered scenarios where the repair results of DroidPalette, Iris, and GPT-4o differed to ensure the survey's validity. Second, we prioritized 10 issue reports with the lowest color contrast, as these issues can have impacts on app users. The survey presented side-by-side screenshots of the original and repaired UIs, each calibrated to be shown in the resolution of the Pixel 5.0 mobile device

used in the experiment. The initial order of the screenshots' placement was randomized and only labeled as Version 1, Version 2, etc., so that respondents would not guess which tool corresponds to each screenshot. Our survey asked respondents to rate their preference for the visual effects of the displayed screenshots using a five-point Likert scale, in order to evaluate the effectiveness of DroidPalette compared to the baselines.

We conducted the survey on Amazon Mechanical Turk (AMT), a crowdsourcing platform widely used for user studies. For AMT participants, we limited feedback collection to individuals over 50 years old, as according to WHO reports, two-thirds of global cases of near-vision impairment occur within this age group [26]–[28]. Additionally, our survey asked participants whether they had any visual impairments. Since our survey was conducted in English, we restricted participation to English-speaking countries. Following the best practices for AMT documented in prior work [29], we implemented strict quality control over both the participants and their responses. We initially selected workers with a past task approval rate above 95% and more than 5,000 approved tasks completed. We also excluded obviously inattentive responses, including: (1) over 95% of answers being the same, (2) answers following a specific pattern, and (3) page viewing times below the minimum limit (<60s per question) or above the maximum limit (>600s). There were 7 such cases in our collected sample. Finally, we collected 55 valid surveys, among which 43 indicated the presence of visual impairments, and 12 reported none.

The user study results are shown in Figure 11. Among responses from participants with visual impairments, 304 entries considered DroidPalette's repairs the best, 214 entries favored Iris, and 142 entries favored GPT-4o. Among responses from participants without visual impairments, 79 entries rated DroidPalette's repairs as the best, 42 entries favored Iris, and 30 entries favored GPT-4o. Due to tied rankings, the total number of votes for these tools may exceed the total number of responses. DroidPalette's repair results were consistently ranked across both groups, coming first in eight questions and second in the remaining two. Specifically, in these two second-place cases, DroidPalette's patches accounted for the colors of other UI components on the page, resulting in slightly lower contrast than the top-ranked repairs. This indicates that, compared to the baseline methods, patches generated by DroidPalette are more likely to be accepted by app users.

Feedback from App Developers. We further explored whether the fixes for color-related accessibility issues in Android apps using DroidPalette can be acceptable by app developers. Different from the issue selection process in our user study, we selected open-source Android apps in F-Droid satisfying the following criteria: (1) with recent update within three months because such projects are actively maintained to receive feedback. (2) DroidPalette is capable of fixing all color-related accessibility issues on the page where the buggy UI component is located. We finally selected 13 open-source apps with color-related accessibility issues from F-droid and applied DroidPalette for the fixes. Following the practices of

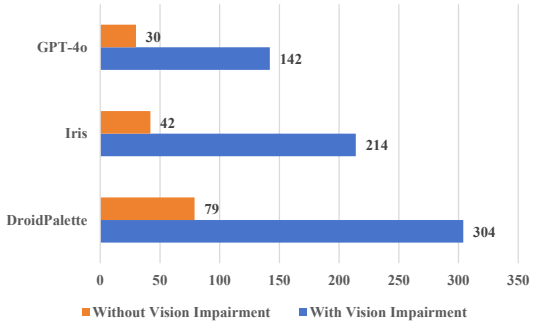


Fig. 11: User study results. The values for the three tools respectively indicate the number of users who selected each repair solution as the most preferred.

Zhang et al. [15], for each app, we filtered out the pages where DroidPalette can fix all color-related accessibility issues within the app's UI page and submitted a Pull Request to its GitHub repository. Through developer feedback, we aim to understand whether DroidPalette meets developers' requirements in terms of issue-fixing color selection and patch generation.

The results are shown in Table II, which details the app name, app category, Pull Request ID and the status of the Pull Request. Of the 13 pull requests we submitted, 8 have received positive feedback from developers, and 4 have been merged by the original app developers. The above results indicate the usefulness of the patches generated by DroidPalette. We received positive feedback from app developers by providing an example as follows:

- "Thanks for the help", in the Pull Request #56 of Pagan [30].

We also received developers feedback from organicmaps [31] that app developers confirmed our issue-fixing strategy but the proposed patch should go through minor revision, because the patch generated by DroidPalette failed to consider adapting to the switching between Android's day and night modes in the app code. We received feedback from the developers of disky [32], who chose not to merge our patch. They explained that the patch we submitted did not adequately adapt to the app's Material You feature, which dynamically adjusts colors based on each device's system settings. To deliver such issue-fixing strategy that app developers expected, analyzing the app's code would be necessary, which is beyond DroidPalette's capabilities. As discussed in Section V, DroidPalette repairs color-related accessibility issues in XML configuration files, given that XML configuration files are the primary carriers of apps' UI, and the majority (71.29%) in our empirical dataset are repaired in this manner. Currently, focusing solely on the XML configuration files has already improved DroidPalette's success rate compared to the baselines. Considering the challenge of understanding the semantics of app code for fault localization and repair, we plan to enhance DroidPalette by incorporating diverse patching strategies to repair color-related accessibility issues in app code.

TABLE II: Feedback from app developers.

App Name	Category	#Issue	Issue State
EnforceDoze [24]	System	#26	Merged
KeePassDroid [33]	Security	#525	Merged
Pagan [30]	System	#56	Merged
RoMote [23]	Multimedia	#49	Merged
Dahdidahdit [34]	Education	#12	Confirmed
Imagepipe [35]	Graphics	#80	Confirmed
Chess [36]	Games	#171	Confirmed
organicmaps [31]	Navigation	#10630	Confirmed
disky [32]	Development	#35	Rejected
solxpect [37]	Science	#32	Pending
SkyTube [38]	Internet	#1360	Pending
seadroid [39]	Internet	#1079	Pending
Simple-Search [40]	Internet	#34	Pending

VII. Threats to Validity

One key limitation of this study is that the participant sampling for our experimental research may not adequately represent populations affected by color perception deficiencies. To mitigate this concern, we specifically recruited 55 individuals aged 50 years and older.

In this study, an internal threat is that users evaluated the UI based only on screenshots, not through direct interaction with the mobile device's UI settings. This could lead to gaps between the simulated usage process and the actual user experience. However, the research design partially addresses this issue. It uses surveys that allow users to rate various repair solutions and derive preference metrics from these ratings.

VIII. Related Work

A. Android Accessibility Issues

The accessibility issues in Android apps are considered one of the main challenges faced by app developers [41]. Existing research has explored this issue from various perspectives. For example, Alshayban et al. [41] analyzed the prevalence, causes, and impact of accessibility issues on user experience from the perspectives of apps, developers, and users, and proposed actionable advice to improve them. Medina et al. [42] conducted a survey analyzing the current state of accessibility issues, and explored future research directions and trends. There are also existing approaches proposed for automatic detection and repair of accessibility issues in both web clients and mobile apps [7], [43]–[49]. For example, Lehmann et al. [45] studied accessibility support for implementing "Total Conversation Service" in Next Generation Networks (NGN), particularly for individuals with hearing, speech, or visual impairments. Ferati et al. [46] emphasized the importance of usage contexts and cultural dimensions in accessibility, pointing out the need for personalized solutions designed according to the specific needs of blind and visually impaired individuals. Eler et al. [47] introduced MATE, an automated tool capable of identifying more accessibility issues than static analysis tools (such as Android Lint) and frameworks that rely on existing test cases (such as Espresso). Alshayban et al. [7] designed AccessiText, a tool focused on detecting text accessibility issues in Android apps, especially in situations where there is incompatibility with text scaling assistive services. Salehnamadi et al. [48] developed Latte, which can reuse UI test cases written by developers to assess the accessibility of apps.

In view of the prevalence of color-related issues, there are existing approaches available to detect and fix them. Specifically, Chen et al. [4] developed Xbot, which integrates Google Accessibility Scanner to detect color-related accessibility issues in Android apps. Zhang et al. [15] introduced Iris, the first automated tool for addressing color-related accessibility issues in Android apps. As illustrated in Section III, existing approaches often fail to generate correct path when the issue-fixing colors are not actually in its referenced database. DroidPalette tackles the above limitation by encoding the app developers' practices during the issue repair process, thus enhancing the success rate on its generated patches.

B. Visual-based Issue Repair

There are existing approaches that focus on repairing issues related to visualization in UI-based software [50]–[55]. Specifically, XFix [53] repairs compatibility issues in layout of web pages. MFix [54] and MobileVisFixer [50] repair mobile-friendly issues in web pages. CBRepair [51] repairs internationalization presentation issues. ConfFix [52] repairs incompatibilities in apps' UI pages. However, the scope of the above approaches goes beyond the automatic repair of color-related accessibility issues. Therefore, this study conducts a study on the repair practices of color-related accessibility issues, and further proposes DroidPalette to automatically repair these issues.

IX. Conclusion

In this paper, we conduct an empirical study on the practices developers use to fix color-related accessibility issues in Android apps. Based on the empirical finding, we further introduce DroidPalette, an automated approach designed to fix color-related accessibility issues in Android apps. Evaluation results indicate that DroidPalette can effectively resolve color accessibility issues while receiving positive feedback from both app users and developers. In the future, we plan to enhance the capability of DroidPalette in repairing color-related accessibility issues by editing the apps' code.

Data Availability

We have released the code of DroidPalette and the empirical study results on GitHub [56], and provided the dataset we used (i.e., 105 APK files) on Google Drive [57].

Acknowledgment

We sincerely thank anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (Grant No. 62402405), Youth Program of the Xiamen Natural Science Foundation (Grant No. 3502Z202471016), and the Fundamental Research Funds for the Central Universities (Grant No. 20720240087).

REFERENCES

[1] 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 738–748.

[2] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[3] I. A. Abdulaziz Alshayban and S. Malek, "Accessibility issues in android apps: state of affairs, sentiments, and ways forward," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1323–1334.

[4] S. Chen, C. Chen, L. Fan, M. Fan, X. Zhan, and Y. Liu, "Accessible or not? an empirical investigation of android app accessibility," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3954–3968, 2022.

[5] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez, "Can everyone use my app? an empirical study on accessibility in android apps," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 41–52.

[6] E. Park, S. Han, H. Bae, R. Kim, S. Lee, D. Lim, and H. Lim, "Development of automatic evaluation tool for mobile accessibility for android application," in *2019 International Conference on Systems of Collaboration Big Data, Internet of Things & Security (SysCoBIoTS)*. IEEE, 2019, pp. 1–6.

[7] A. Alshayban and S. Malek, "Accessitext: automated detection of text accessibility issues in android apps," in *2022 ACM/IEEE 30th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. IEEE, 2022, p. 984–995.

[8] Amnet, "Ensuring mobile accessibility: Color contrast," Apr 2021, posted on April 12, 2021. [Online]. Available: https://amnet-systems.com/ensuring-mobile-accessibility-color-contrast/

[9] L. Zhou, V. Bansal, and D. Zhang, "Color adaptation for improving mobile web accessibility," in *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*. IEEE, 2014, pp. 291–296.

[10] F. E. Sandnes, "Inverse color contrast checker: Automatically suggesting color adjustments that meet contrast requirements on the web," in *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, 2021, p. 4.

[11] OneBusAway, "Onebusaway android." [Online]. Available: https://github.com/OneBusAway/onebusaway-android/pull/1126

[12] H. Huang, M. Wen, L. Wei, Y. Liu, and S.-C. Cheung, "Characterizing and detecting configuration compatibility issues in android apps," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 517–528.

[13] Google, "Accessibility scanner," 2022, accessed: 2025-02-17. [Online]. Available: https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_SG

[14] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "Multi-objective optimization of energy consumption of guis in android apps," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 1–47, 2018.

[15] Y. Zhang, S. Chen, L. Fan, C. Chen, and X. Li, "Automated and context-aware repair of color-related accessibility issues for android apps," in *2023 ACM/IEEE 31st Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 1255–1267.

[16] F. E. Sandnes, "Inverse color contrast checker: automatically suggesting color adjustments that meet contrast requirements on the web," in *2021 International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS)*, 2021, pp. 1–4.

[17] W3C, "Text or image contrast," 2022, accessed: 2025-03-02. [Online]. Available: https://www.w3.org/WAI/WCAG21/quickref/?versions=2.0#contrast-minimum

[18] openai, "Gpt-4o," https://openai.com/index/gpt-4o.

[19] v4lpt, "vpc," https://github.com/v4lpt/VPC.

[20] wordpress mobile, "Wordpress-android," https://github.com/wordpress-mobile/WordPress-Android/pull/14849.

[21] breezy weather, "breezy-weather," https://github.com/breezy-weather/breezy-weather/pull/938.

[22] "Write automated tests with ui automator," 2023. [Online]. Available: https://developer.android.com/training/testing/other-components/ui-automator

[23] wseemann, "Romote," https://github.com/wseemann/RoMote.

[24] farfromrefug, "Enforcedoze," https://github.com/farfromrefug/EnforceDoze.

[25] nilsbraden, "ttrss-reader-fork," https://github.com/nilsbraden/ttrss-reader-fork.

[26] K. D. Frick, S. M. Joy, D. A. Wilson, K. S. Naidoo, and B. A. Holden, "The global burden of potential productivity loss from uncorrected presbyopia," in *Ophthalmology*, 2015, pp. 1706–1710.

[27] "Universal eye health: A global action plan 2014-2019," 2013. [Online]. Available: https://www.who.int/publications/i/item/universal-eye-health-a-global-action-plan-2014-2019

[28] "World report on vision." 2019. [Online]. Available: https://www.who.int/docs/default-source/documents/publications/world-vision-report-accessible.pdf

[29] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond, "Automated repair of size-based inaccessibility issues in mobile applications," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 730–742.

[30] quintinfsmith, "Pagan," https://github.com/quintinfsmith/pagan.

[31] organicmaps, "organicmaps," https://github.com/organicmaps/organicmaps.

[32] newhinton, "disky," https://github.com/newhinton/disky.

[33] bpellin, "Keepassdroid," https://github.com/bpellin/keepassdroid.

[34] matthiasjordan, "Dahdidahdit," https://f-droid.org/zh_Hans/packages/com.paddlesandbugs.dahdidahdit/.

[35] Starfish, "Imagepipe," https://codeberg.org/Starfish/Imagepipe.

[36] jcarolus, "Chess," https://github.com/jcarolus/android-chess.

[37] woheller69, "solxpect," https://github.com/woheller69/solxpect.

[38] SkyTubeTeam, "Skytube," https://github.com/SkyTubeTeam/SkyTube.

[39] haiwen, "seadroid," https://github.com/haiwen/seadroid.

[40] TobiasBielefeld, "Simple-search," https://github.com/TobiasBielefeld/Simple-Search.

[41] A. Alshayban, I. Ahmed, and S. Malek, "Accessibility issues in android apps: State of affairs, sentiments, and ways forward," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1323–1334.

[42] L. Medina, G. Lopez, I. Díaz-Oreiro, and J. A. Brenes, "Current practices in accessibility evaluation: A literature review of over 100 studies," in *2024 IEEE VII Congreso Internacional en Inteligencia Ambiental, Ingeniería de Software y Salud Electrónica y Móvil (AmITIC)*. IEEE, 2024, pp. 1–8.

[43] H. Huang, L. Wei, Y. Liu, and S.-C. Cheung, "Understanding and detecting callback compatibility issues for android applications," in *2018 IEEE/ACM 33rd International Conference on Automated Software Engineering (ASE)*, 2018, pp. 532–542.

[44] L. Wei, Y. Liu, S.-C. Cheung, H. Huang, X. Lu, and X. Liu, "Understanding and detecting fragmentation-induced compatibility issues for android apps," *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1176–1199, 2020.

[45] L. Lehmann, "Accessibility support for persons with disabilities by total conversation service mobility management in next generation networks," in *Proceedings of ITU Kaleidoscope 2011: The Fully Networked Human? - Innovations for Future Networks and Services (K-2011)*. IEEE, 2011, pp. 1–7.

[46] M. Ferati, B. Raufi, A. Kurti, and B. Vogel, "Accessibility requirements for blind and visually impaired in a regional context: An exploratory study," in *2014 IEEE 2nd International Workshop on Usability and Accessibility Focused Requirements Engineering (UsARE)*. IEEE, 2014, pp. 13–16.

[47] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, "Automated accessibility testing of mobile apps," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 116–126.

[48] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek, "Latte: Use-case and assistive-service driven automated accessibility testing framework for android," in *2021 CHI Conference on Human Factors in Computing Systems (CHI)*, 2021, pp. 1–11.

[49] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, "A tale of two cities: How webview induces bugs to android applications," in *2018 Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 702–713.

[50] A. Wu, W. Tong, T. Dwyer, B. Lee, P. Isenberg, and H. Qu, "Mobilevisfixer: Tailoring web visualizations for mobile phones leveraging an explainable reinforcement learning framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 464–474, 2021.

[51] A. Alameer, P. T. Chiou, and W. G. Halfond, "Efficiently repairing internationalization presentation failures by solving layout constraints," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 172–182.

[52] H. Huang, C. Xu, M. Wen, Y. Liu, and S.-C. Cheung, "Conffix: Repairing configuration compatibility issues in android apps," in *2023 ACM SIGSOFT 32nd International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 514–525.

[53] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond, "Automated repair of layout cross browser issues using search-based techniques," in *2017 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. IEEE, 2017, pp. 249–260.

[54] ——, "Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 215–226.

[55] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang, "Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 322–334.

[56] DroidPalette, "Droidpalette," https://github.com/good-good-create/DroidPalette.

[57] DroidPalette, "dataset." [Online]. Available: https://drive.google.com/file/d/136trbD78Z0P0r4b5OZp1xzgCbz-ZdGKp/view?usp=drive_link