Scenario-Driven and Context-Aware Automated Accessibility Testing for Android Apps

Yuxin Zhang*, Sen Chen†§, Xiaofei Xie‡, Zibo Liu*, and Lingling Fan†
*College of Intelligence and Computing, Tianjin University, Tianjin, China
†College of Cyber Science, Nankai University, Tianjin, China
‡Singapore Management University, Singapore, Singapore

***Abstract*—Mobile accessibility is increasingly important nowadays as it enables people with disabilities to use mobile applications to perform daily tasks. Ensuring mobile accessibility not only benefits those with disabilities but also enhances the user experience for all users, making applications more intuitive and user-friendly. Although numerous tools are available for testing and detecting accessibility issues in Android applications, a large number of false negatives and false positives persist due to limitations in the existing approaches, i.e., low coverage of UI scenarios and lack of consideration of runtime context. To address these problems, in this paper, we propose a scenario-driven exploration method for improving the coverage of UI scenarios, thereby detecting accessibility issues within the application, and ultimately reducing false negatives. Furthermore, to reduce false positives caused by not considering the runtime context, we propose a context-aware detection method that provides a more fine-grained detection capability.**

Our experimental results reveal that A11yScan can detect 1.7X more issues surpassing current state-of-the-art approaches like Xbot (3,991 vs. 2,321), thereby reducing the false negative rate by 41.84%. Additionally, it outperforms established UI exploration techniques such as SceneDroid (952 vs. 661 UI scenarios), while achieving comparable activity coverage to recent leading GUI testing tools like GPTDroid on the available dataset (73% vs. 71%). Meanwhile, with the context-aware detection method, A11yScan effectively reduces the false positive rate by 21%, validated with a 90.56% accuracy rate through a user study.

***Index Terms*—Mobile accessibility, Accessibility testing, Android app, UI exploration, Context-aware analysis**

I. Introduction

The increasing popularity of mobile applications (apps) has made them an essential element in people's daily lives, and mobile apps have become an important medium for accessing information and services. However, many of these apps are inaccessible to people with disabilities, making it difficult for them to use and enjoy the same benefits as non-disabled users [1]–[4]. The World Health Organization (WHO) estimates that approximately 15% of the world's population has a disability [5], which means that app developers must ensure that all users, including disabled users, can access their apps [6]–[9]. Despite increasing awareness of accessibility, a large number of developers still struggle to create accessible apps due to a lack of knowledge or understanding of accessibility guidelines [10]. Thus, effective tools are urgently needed to help developers detect accessibility issues in their apps.

§ Sen Chen is the corresponding author (Email: tigersenchen@163.com).

In recent years, researchers have developed many tools to test and detect accessibility issues in apps and conduct empirical research on these issues [3], [11], [12]. These tools are mainly divided into two categories: static and dynamic analysis tools. Static analysis tools [13]–[15] such as Android Lint [16] detect accessibility issues by analyzing static information such as code and resource files without running the apps, however, they are ineffective and time-consuming in detecting mobile accessibility issues [3], [7]. Dynamic analysis tools, on the other hand, detect accessibility issues by running the app with different UI exploration methods, such as manually dynamic methods [17]–[20], script-based dynamic methods [21]–[23], and automated dynamic methods [7], [24]. Since the issue coverage of accessibility detection depends on the number of explored UI pages, their common goal is to reach more **UI scenarios** and check whether the attributes of the UI components in them violate accessibility standards. App UI scenarios include *Android Activity itself, Activity-dependent, and Activity-sensitive UI scenarios.* Each Activity includes various UI scenarios beyond the initially rendered UI page during runtime, such as fragment UI, drawer, menu, and dialog, which refer to Activity-dependent UI scenarios. Activity-sensitive UI scenarios refer to the new UI states of the current Activity when the states of other UI scenarios change due to user interactions. As shown in Fig. 1, clicking the menu button in the upper right corner of the first *MainActivity* can reach an Activity-dependent UI scenario (i.e., the second updated *MainActivity*), while user interaction flows (Adding a new category in *CategoriesActivity*) can lead to an activity-sensitive UI scenario (i.e., the last updated *MainActivity*). Failure to explore activity-dependent or activity-sensitive UI scenarios of apps would result in the inability to detect accessibility issues in them, such as Issue1, Issue2, and Issue3 in this example. However, existing accessibility testing tools have some significant drawbacks: **(1) Low coverage of UI scenarios causing false negatives.** The existing manual and script-based tools can only cover limited UI scenarios, and the current automated dynamic accessibility testing tools primarily focus on detecting the accessibility of activities, while overlooking the accessibility issues that may arise from activity-dependent and activity-sensitive UI scenarios. In addition to accessibility detection tools, app GUI testing [25]–[34] and UI exploration [35]–[38] tools also constantly strive to

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00093

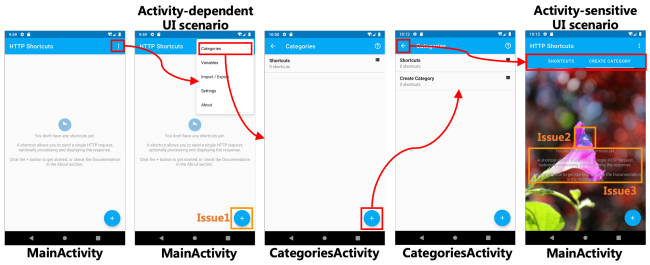

Fig. 1: New accessibility issues detected in activity-dependent and activity-sensitive UI scenarios.

extract more UI scenarios for different purposes, which would enhance the ability of accessibility testing. Although these tools cover a wide range of UI components and paths to improve activity/UI coverage, their main goal is to improve coverage or identify general bugs, often ignoring the different attributes of the same UI component in different UI scenarios, which are crucial for accessibility testing as they can determine whether the UI component meets accessibility standards. Ignoring this difference can lead to missed detection of a component's accessibility issues in specific scenarios. **(2) Lack of consideration of runtime context causing false positives.** The existing accessibility testing tools directly rely on accessibility rules [39] to check whether the attributes of the detected component violate the accessibility standards, and lack consideration of the actual runtime context of the detected component, which can lead to incorrect judgments about the attributes of the UI component in different usage scenarios, resulting in false positives. For example, these tools may mistakenly identify *TextView* components that have been set as invisible by developer as low contrast issues. Whether a UI component has accessibility issues requires not only checking whether its current attributes violate accessibility rules but also considering contextual information during app runtime, such as the hierarchy of all UI components in the current UI page and the state of this UI component itself (i.e., (in)visible, (un)filled, and (un)checked).

To address these above problems, we propose A11yScan, an automated approach for accurately detecting accessibility issues in Android apps. Specifically, to solve the first problem, we design a scenario-driven UI exploration method that dynamically explores as many UI scenarios as possible of the apps, to expand the scope of accessibility detection using two key strategies: (1) initial state exploration, which uses the command-line activity launches combined with a depth-first exploration strategy featuring multi-state combination to interact with the interactive UI components of each Activity to increase activity coverage while simultaneously acquiring additional activity-dependent UI scenarios and the various attributes of UI components in them, and (2) enhanced state re-exploration, which analyzes and re-explores critical UI scenarios with two or more out-degrees in the UI scenario transitions generated from initial state exploration to identify activity-sensitive UI scenarios and the attribute changes of the same UI component in new detected UI scenarios. Through high UI scenario coverage, the scenario-driven UI exploration method targets accessibility testing by capturing the different attributes of UI components across different UI scenarios, ensuring comprehensive accessibility assessment of the app. To solve the second problem, we propose a context-aware detection method that designs newly-defined checking rules, referred to as oracle, by combining the pre-defined rules of testing frameworks like Accessibility Testing Frameworks (ATF) [39] and the contextual information gained from constructing resource trees of the UI components. These new rules in this oracle are not about redefining accessibility standards but optimizing the detection implementations to make results more accurate and aligned with standards. Additionally, the extraction of contextual information is independent of specific issue types. While these designs are systematic, the checking rules still need to be fine-tuned according to different issue types, ensuring accurate accessibility assessment of the app.

We conducted a series of experiments on 100 real-world Android apps, including both closed-source and open-source apps, to evaluate the effectiveness and efficiency of A11yScan. The experimental results indicate that A11yScan performs well in issue detection with 1.7X more issues (Reduced the false negative rate by 41.84%) compared to the state-of-the-art accessibility testing tool (i.e., Xbot [7]). Furthermore, A11yScan outperforms in extracting UI scenarios compared with the state-of-the-art UI exploration tools like SceneDroid (952 vs. 661 UI scenarios). Additionally, to ensure the comprehensiveness of effectiveness evaluation, we further take into account the recent state-of-the-art GUI testing tool (i.e., GPTDroid [31]). We compared it using the metric of activity coverage and app dataset used in their paper rather than the number of explored UI scenarios and detected issues due to the unavailability of their tool. A11yScan achieved a comparable result on activity coverage (73% vs. 71%), which demonstrates that the ability of the scenario-driven exploration method is comparable even compared with the cutting-edge technique-enhanced approach with the large language models.

Additionally, false positives, identified through context-aware issue detection, represent 21% of detected issues. Subsequent user study verified an accuracy of 90.56%, highlighting the effectiveness of A11yScan in handling false positives. The average time for testing one app is 15.16 minutes.

In summary, our main contributions are as follows.

- We are the first work to attempt to address the significant problem of false alarms when testing app accessibility.
- We proposed a scenario-driven UI exploration method, aiming to explore both activity-dependent and activity-sensitive UI scenarios to find more accessibility issues.
- We introduced a context-aware detection method aimed at reducing false positives by leveraging contextual information from UI components during the app's runtime.
- We have released A11yScan as well as the used dataset on https://github.com/tjuyuxinzhang/A11yScan-mobile.git.

II. Background

A. UI scenarios of Android Apps

UI scenario is crucial for testing Android apps. The UI scenario covers a series of state changes during the interaction between users and apps, reflecting the dynamics and interactivity of the software. The following will explore three key types of UI scenarios: *Activity, Activity-dependent UI scenarios, and Activity-sensitive UI scenarios.*

Activity is a fundamental component in Android apps, representing a single screen with a UI scenario. It serves not only as a container for user interactions but also as the primary means for the app to present information.

Activity-dependent UI scenarios involve state changes triggered by user interaction with UI components such as *EditText* and *Button* within a single Activity. These changes usually occur after users perform operations such as clicking buttons, selecting drop-down menu items, or entering data. The state changes in such scenarios are predictable as they directly depend on user interactions. As shown in Fig. 1, when the user clicks on the menu button in the first *MainActivity* (circled in red), a drawer is triggered, which generates a new activity-dependent UI scenario (i.e., the second updated *MainActivity*).

Activity-sensitive UI scenarios focus on UI state changes caused by user interactions, which may indirectly affect the current or related activities. The changes in these scenarios are indirect and may not immediately manifest, as they depend on the operations accumulated by users during the use of the app. For example, a user's actions in one Activity may affect the data display of another Activity, even if there is no direct interaction path between the two. In the subsequent section of Fig. 1, when the user creates a new category in *CategoriesActvity*, the *MainActivity* is updated and expanded, which refers to an activity-sensitive UI scenario (i.e., the last updated *MainActivity*).

B. Accessibility Standards

Mobile accessibility is an important aspect of ensuring equal access to technology for people with disabilities. To improve the accessibility of mobile devices, many accessibility standards and developer guidelines have been proposed, including those from W3C, Web Content Accessibility Guidelines (WCAG) 2.0 [40] and 2.1 [41], Google Accessibility Guidelines for Android [42], ISO 9241 developed by the International Organization for Standardization (ISO) [43], the US Revised Section 508 Standards [44], and the BBC Mobile Accessibility Standards [45] and Guidelines from the UK [46], among others. These standards provide recommendations to better support people with different types of disabilities, including those with mobility, hearing, and visual impairments.

Besides these standards and guidelines, companies have developed their own accessibility guidelines based on industry standards. For example, the Android Accessibility Developer Guidelines [47], Apple Accessibility Developer Guidelines [48], and IBM Accessibility Checklist [19] provide developers with specific recommendations and best practices for creating accessible mobile apps.

TABLE I: List of accessibility testing tools for Android apps.

Category	Strategy	Tools
Static Analysis	Automated Static Method	Lint [16], PMD [13], Checkstyle [14]
Dynamic Analysis	Manual Dynamic Method	Accessibility Scanner [17], [18], IBM AbilityLab Mobile Accessibility Checker [19]
		TalkBack [53], Switch Access [54], A11yPuppetry [20]
	Script-based Dynamic Method	Espresso [22], Robotium [55], Robolectric [56]
		Latte [23], UIautomator [21], Appium [57]
	Automated Dynamic Method	MATE [58], Groundhog [24], PUMA [59], forApp [60]
		Xbot [7]

There are also some relevant laws and regulations. For example, the US Americans with Disabilities Act (ADA) [49] and the Communications and Video Accessibility Act [50] require federal agencies to use technology that meets 508 standards [44] to ensure that people with disabilities can use these technologies. Similarly, the European Union has also issued the Convention on the Rights of Persons with Disabilities [51] and requires member states to take measures to ensure that people with disabilities can access digital technologies [52] such as mobile devices and the Internet.

C. Accessibility Testing Tools

Table I lists the commonly-used accessibility testing tools for Android apps, which can be classified into two categories based on their used techniques: static and dynamic analysis.

1) Static Analysis: Static analysis can detect accessibility issues by analyzing the source code or compiled code of the app. However, this type of tool often fails to capture accessibility issues that only appear at runtime and cannot consider the actual runtime environment of the app. Tools like Lint [16], PDM [13], and Checkstyle [14] can be used to detect errors and potential issues in the code, such as missing content descriptions and accessibility tags in XML layout files, which can improve code quality.

2) Dynamic Analysis: Dynamic analysis can detect accessibility issues at runtime to test the accessibility of the app. This approach overcomes the limitations of static analysis and can more accurately simulate how real users use the app. Dynamic analysis tools can be classified into three types based on their exploration strategy: *manual dynamic method*, *script-based dynamic method*, and *automated dynamic method.*

a) Manual dynamic method: This method requires developers to manually simulate user interactions and check the accessibility of the app. Therefore, it requires developers to have a certain level of knowledge and skills in accessibility, and is relatively time-consuming and labor-intensive, making it less suitable for large-scale app testing. By interacting with the components on the UI page manually, developers can visually determine whether the detected issues are true or not. It can also effectively explore more complex components in some scenarios, but reproducing issues may be

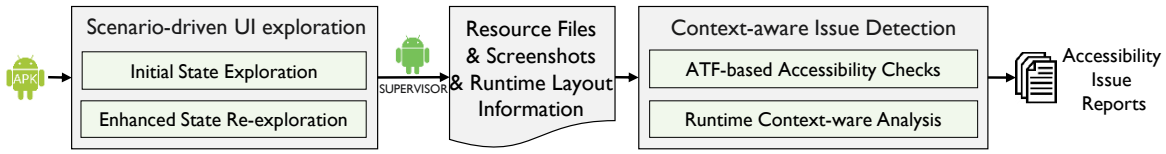

Fig. 2: Overview of A11yScan.

more difficult. Currently, commonly-used tools such as the Accessibility Scanner [17]. It can automatically scan while manually exploring the UI pages, and provide detailed reports on accessibility issues and repair suggestions. In addition, the Android system provides some assistive services to help the disabled use Android devices, such as TalkBack [53] for visual impairments and Switch Access [54] for motor impairments. These tools can assist testers in simulating the behavior of different users when using the app.

b) Script-based dynamic method: This method uses scripts to simulate user interactions and check the accessibility of the app. It is more automated and efficient than the manual dynamic and can be used for large-scale app testing. These scripts can be customized for different scenarios and can be easily executed repeatedly. However, writing suitable scripts is a challenge. Commonly used tools include Espresso [22] and Robotium. In addition, others such as Latte [23], Appium [57], and UIautomator [21] can integrate assistive services for testing and focus on meeting the needs of different users.

c) Automated dynamic method: This method can fully automatically simulate user interactions and check the accessibility of the app. It is the highest degree of automation, and can quickly and accurately detect accessibility issues in large and complex apps, making it suitable for large-scale app testing. Existing tools attempt to explore UI pages using different methods to discover potential issues on more UI pages. For example, both MATE [58] and Groundhog [24] use the Monkey [61], [62] to simulate user interactions with the app, whose random strategy can lead to incomplete exploration. Xbot [7] uses instrumentation technique and static data-flow analysis based on Activity intent parameters to explore UI pages, achieving a higher activity coverage, but ignoring activity-dependent and activity-sensitive UI scenarios triggered by interactive components as well as user interactions.

III. Approach

To overcome the limitations of existing state-of-the-art approaches in terms of false negatives and false positives, we propose a fully automated tool for testing and detecting accessibility issues in Android apps, named A11yScan, which takes an APK file as input and outputs detection reports for accessibility issues and other relevant parsing results. As shown in Fig. 2, A11yScan mainly consists of two phases: *(1) Scenario-driven UI Exploration*, which thoroughly explores more UI scenarios of the app to increase scenario coverage and detection range, to discover more potential accessibility issues. *(2) Context-aware Issue Detection*, which checks the accessibility of the detected UI components by newly-defined checking rules, combining the pre-defined rules of current accessibility testing frameworks and the contextual information of detected UI components by constructing resource trees of the relevant UI components.

A. Scenario-driven UI Exploration

To minimize the occurrence of false negatives, our goal is to capture as many UI scenarios as possible in the apps. However, this is a challenging task due to various limitations of existing tools. (1) Firstly, the complex code structure and diverse scenario designs of apps make it difficult to simulate a wide range of user interactions and consider the impact of user interactions between different UI components, thus limiting the ability to explore activities and more UI scenarios with multiple UI states. (2) Secondly, existing tools lack consideration of the UI state changes generated after complex user interactions, which may accompany attribute changes of UI components and thus fall short of identifying and exploring activity-sensitive UI scenarios. To achieve higher coverage of UI scenarios, we have adopted two key strategies:

1) Initial State Exploration: Android provides an interface to directly launch activities from the console using the Android Debug Bridge commands. This involves using Intent objects to specify the target Activity and pass necessary data, where Intent serves as one of the mechanisms for inter-component communication (ICC) in Android, supporting message exchange and operation execution. To generate Intent objects, A11yScan parses the AndroidManifest.xml file and Java code to obtain the basic attributes (including action, category, data, and type) and extra parameters (composed of basic structures such as String, Char, and Boolean) of Intent objects. And fill in extra parameters based on data types. A11yScan then uses these Intent objects to directly launch the Activity from the console for future testing tasks like Fax [63] and Story-Distiller [37]. This establishes the foundation for exploring various UI scenarios by adopting different strategies.

In Android apps, each UI page is made up of various UI components, ranging from simple buttons and text fields to more complex image carousels and navigation menus [64]. User interactions with these UI components trigger new UI scenarios and interactions. For example, clicking a *Button* component may open a new UI page or display additional information. Inputting text or numbers into an *EditText* component may guide the user to different pages or sections of the app. Therefore, after launching a new UI page, A11yScan retrieves the interactable UI components on the current page and employs a top-down depth-first exploration strategy to interact with each component individually, such as filling *EditText* components with the randomly generated inputs according to the attribute *inputType* of *EditText* and clicking on clickable

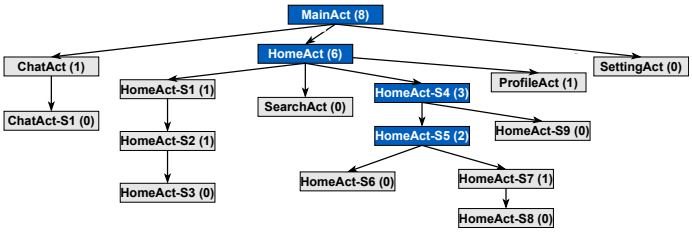

Fig. 3: UI scenario transitions generated by initial state exploration. The node represents UI scenarios and (#) refers to their out-degrees.

components. Additionally, A11yScan evaluates all possible combinations of component states to capture the diversity and dynamics of real user interactions. For example, with an *EditText* (filled/blank), a *CheckBox* (checked/unchecked), and a *Switch Button* (on/off), it explores all $2^3 = 8$ initial UI states through multi-state combination. Then A11yScan continues depth-first exploration until all possible UI scenarios are explored or no new scenarios are found. Additionally, scrollable pages will be fully presented through scrolling.

Overall, A11yScan utilizes command-line activity launches followed by a top-down depth-first exploration strategy to trigger a wider variety of component states and UI states, effectively reaching more activities and exploring activity-dependent UI scenarios and capturing various attributes of UI components in them.

2) Enhanced State Re-exploration: After the initial UI exploration, some UI scenarios might change due to user interactions. These changes often involve activity-sensitive UI scenarios, where the UI state dynamically changes with the operations accumulated by users. To accurately capture these changes, we developed an advanced strategy to identify these activity-sensitive UI scenarios and detect whether there are new UI scenarios triggered by previous exploration behaviors.

A11yScan identifies critical UI scenarios that are susceptible to state changes induced by interactions in other scenarios by analyzing UI scenario transitions following the initial state exploration. Fig. 3 represents the UI scenario transitions generated after the initial state exploration of a real app, showing the transition path between activities and other UI scenarios. Each node represents a UI scenario including Activity and Activity-dependent scenarios, and (#) on them refers to their out-degrees, reflecting their significance in the app logic and their sensitivity to state changes. For example, *MainAct* can reach 8 event flows of UI scenario, indicating that it carries more functions, while *ProfileAct* can only reach one with simpler logic. If a UI scenario can reach multiple paths, we consider it a critical UI scenario, which is sensitive to changes in the UI state, such as *MainAct*, *HomeAct*, *HomeAct-S4*, and *HomeAct-S5* in Fig. 3. Subsequently, A11yScan employs a bottom-up depth-first exploration approach to re-explore the marked critical UI scenarios to identify activity-sensitive UI scenarios as well as the attribute changes of the same UI component in new detected UI scenarios. This bottom-up strategy, distinct from the top-down used during the initial state exploration, simulates potential random user interactions. This exploration

Algorithm 1: Scenario-driven UI Exploration

Input: act_{all}: All activities with Intent objects in the app.
Output: S: All UI scenarios explored within the app.

```
1  foreach act, Intent ∈ act_all do
2      S ← S ∪ startAct(act, Intent)
3      S ← S ∪ InitialExploration(act)
4  S ← S ∪ EnhancedReExploration(S)
5  return S
6  Function InitialExploration(act):
7      Comps_int ← GetInteractableComps(act)
8      foreach comp ∈ Comps_int do
9          S ← S ∪ InteractWithComp(comp)
10     return S
11 Function EnhancedReExploration(S):
12     S_key ← IdentifyKeyScenarios(S)
13     foreach sk ∈ S_key do
14         if DetectStateChange(sk) then
15             S ← S ∪ ReExploreScenario(sk)
16     return S
```

aims to verify if our initial actions have triggered new UI states or behaviors, revealing more complex and dynamic UI scenarios.

It is worth noting that A11yScan dumps the layout structure of the UI page runtime and extracts the resource-id, class, and package attributes of each UI component. It uses the MD5 hash algorithm [65] to generate unique identifiers for each UI component, thereby identifying and distinguishing unique scenarios and avoiding the exploration of duplicate scenarios.

Algorithm 1 delineates the entire process of scenario-driven UI exploration, which employs two exploration strategies. The input is all activities with associated Intent objects in the app, and the output consists of UI scenarios explored using exploration. Specifically, each Activity act is directly started using Intent objects $Intent$ (Lines 1∼2), followed by an initial state exploration through the method InitialExploration (Line 3). During initial state exploration (Lines 6∼10), all runtime interactable components $Comps_{int}$ are first retrieved (Line 7), and a top-down interaction with these components is conducted to explore all reachable activity-dependent UI scenarios (Lines 8∼9). After the initial exploration, an enhanced state re-exploration of the obtained UI scenarios is performed by the method EnhancedReExploration (Line 4) to capture activity-sensitive UI scenarios. In the re-exploration (Lines 11∼16), the UI scenario transitions generated after the initial exploration are analyzed to identify critical UI scenarios S_{key} (Line 12). Each critical UI scenario sk is then checked for state changes resulting from interactions during the initial exploration (Lines 13∼14); if changes are detected, indicating an activity-sensitive UI scenario, a bottom-up re-exploration of that scenario is conducted (Line 15). Finally, upon completion of all explorations, the collection of UI scenarios, S, is returned.

B. Context-aware Issue Detection

Rule-based checks are pre-defined methods based on accessibility standards, making it difficult to consider the diversity of the runtime environment and the states of the detected components. Existing tools use UIAutomator [66] to capture UI layouts and screenshots to obtain UI component attributes for accessibility detection, which can be affected by incorrect screenshots or missing context, leading to existing rule-based misjudgments. To make the detection issues more comprehensive and reliable, we newly defined the checking rules by combining the pre-defined rules of the official accessibility testing framework ATF [39] developed and maintained by Google, and the runtime contextual information of detected UI components.

1) ATF-based Accessibility Checks: To help app developers test the accessibility of their apps, Google provides an Accessibility Testing Framework (ATF) [39] commonly used in accessibility testing tools, which includes a set of APIs for testing the accessibility of apps and websites and checking compliance with accessibility design guidelines such as WCAG [40], [41]. However, ATF itself is not a standalone testing tool. To use ATF, developers need to integrate it into their existing app development and testing processes.

For the standardization of accessibility detection, we still use pre-defined ATF as part of our newly defined rules, which follows the principles of perceivable, operable, understandable, and robust, enabling it to detect accessibility issues related to color contrast, touch size, accessibility labels, and compatibility with screen readers, among others [40], [41]. With the Accessibility Events from the View of the detected UI scenario as input, the Accessibility Testing Framework (ATF) can perform pre-defined accessibility checks on all components of the current UI scenario, which means that ATF can analyze every component in the UI, including buttons, text boxes, labels, etc., to ensure they comply with pre-defined accessibility standards and rules.

2) Runtime Context-aware Analysis: Although integrating ATF can help us detect accessibility issues in the apps, it is not a perfect solution. ATF only performs accessibility checks on input Accessibility Events using pre-defined methods, which may lead to potential risks. These Accessibility Events refer to static data of the captured attributes of UI components and their UI screenshots. Lack of contextual analysis can easily lead to inaccurate judgments of attributes of UI components, especially for some visual design issues or complex interaction scenarios. Here are some factors that could lead to false positives without contextual information:

- *External Dependency.* External Dependency issues arise when apps display third-party services or images with significant visual content such as logos or brand names from other apps during runtime. Accessibility standards do not require contrast checks for these elements or for invalid issues like unused components [41]. Detection without contextual information cannot recognize and categorize the types and uses of detected UI components, leading to misjudgments that violate the definition of accessibility standards.[1] FP (External Dependency) in Fig. 4 shows the circled component was reported as an Image Contrast issue, but it actually directly references a third-party app's icon without contrast requirement in the detected app.
- *UI Presentation.* ATF detects accessibility issues according to the captured attributes of UI components and their UI screenshots. If the UI components are not displayed or not configured correctly, the detection tool may incorrectly indicate accessibility issues. FP (UI Presentation) in Fig. 4 shows the *TextView* component in the app was not displayed correctly due to occlusion, resulting in ATF falsely reporting the component has low contrast issue. It is worth noting that such false positives mainly occur in components that rely on graphical presentation, where incomplete display results in inconsistencies between the detected issues and the actual attributes.
- *Specific UI Design.* Specific UI design refers to intentional UI decisions made by app developers to improve the overall user experience, such as intentionally weakening a component to make it less visible or even invisible in the UI. These designs often have multiple interactive states, such as the filled/blank states of an *EditText* component or the selected/unselected states of a navigation bar. Despite appearing to violate accessibility standards, our investigation reveals most users/app developers view these designs as improvements to the overall user experience rather than accessibility issues. However, ATF does not consider the specific context of the component, making it challenging to identify these specific UI designs, such as the hint text in the *EditText* component (FP (UI Design of EditText)) and the selected status of the navigation bar (FP (UI Design of Navigation)) in Fig. 4. Unlike the previous two types of false positives that violate the definition of accessibility standards, these special UI designs are often contentious regarding their accessibility impact. A11yScan recognizes these special UI designs, categorizes, and annotates them in issue reports, leaving the decision of fixing them to the app owners.

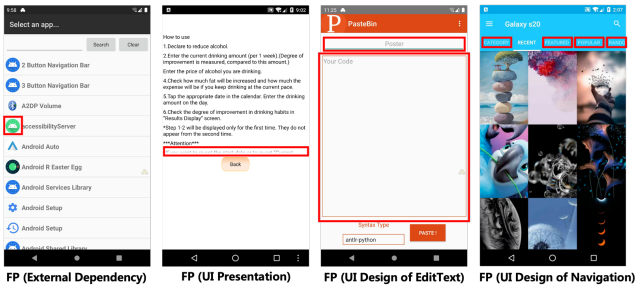

Fig. 4: Examples of false positives under different scenarios.

Therefore, when using ATF to check the accessibility of

[1]Text or images of text that are part of an inactive user interface component, that are pure decoration, that are not visible to anyone, or that are part of a picture that contains significant other visual content, have no contrast requirement. Text that is part of a logo or brand name has no minimum contrast requirement [40], [41].

apps, it is necessary to combine the contextual information of detected UI components (such as their size, visibility, color settings, and their relationship with surrounding UI components) to assist in issue confirmation. To overcome the limitations of the testing framework, A11yScan has integrated a runtime context-aware analysis module that constructs two types of resource trees to provide runtime contextual information for detecting app accessibility issues. The pre-defined rules of ATF and the contextual information gained from constructing resource trees together constitute the newly-defined checking rules. These resource trees represent the hierarchical structure of UI components or views in a UI scenario within an app, with each component node having rich attributes such as size, color, visibility, etc.

The first type of resource tree is built based on the decompiled resource files of the app, providing a comprehensive view of the static structure of the app. This tree reflects the expected structure and organization of the UI designed by developers, including the hierarchical structure and pre-defined attributes of all static elements such as Button, TextView, and Image, typically defined in XML layout files. When using this type of resource tree, we aim to get the attributes of the detected UI component and its structural relationship with surrounding components. Thus, we build the resource tree based on UI component groups in each decompiled XML file, without needing a complete predefined XML file.

The second type of resource tree is built based on the actual runtime layout of the app, reflecting its dynamic performance. This tree includes dynamically generated or modified UI components and their runtime attributes, used to capture the real-time state and behavior of the UI page. Additionally, for components defined in the code, which lack representation in the first type of resource tree, we also extract the necessary contextual information by analyzing the relationships between UI components in the actual runtime layout.

These two types of resource trees not only consider the pre-defined static design and UI component attributes of the app but also the possible changes that may occur during user interactions. Their combination can provide rich contextual information and make up for the shortcomings of ATF for accessibility detection. This method enables us to handle different UI implementations and ensure accurate extraction of relevant information. Using the resource trees, the following checks were mainly verified:

- *External Dependency and Invalid issues.* To check for external dependencies or invalid issues in the detected components, A11yScan extracts attributes of each node in the resource trees as contextual information. The package name, image path, Text, size, etc. of the UI component are important contextual information for accessibility detection. For example, if the package name is inconsistent with the current app, it may be components of other apps or system components that do not belong to the detected app. Images can be obtained from the image path for contrast checking. *text*="" at runtime sometimes means invisible, etc. Through these attributes, it can be determined whether the components presented in the app depend on external resources, such as images from external links, and prevent unused components (i.e., (in)visible, (un)filled, and (un)checked) from being detected by checking their attributes.
- *UI Presentation issues.* To check the status of dynamic loading and collapsing of UI pages, A11yScan identifies incomplete displayed or missing UI components by comparing the differences between the static resource tree and the dynamic resource tree, specifically those defined in the static resource tree but absent in the dynamic resource tree. At this point, the layout information about the detected component in the static resource tree can predict its approximate position at runtime. If it is found in the same position in the dynamic resource tree but has different attributes in two resource trees, it may not be fully displayed. In addition, for dynamically loaded components that are not in the static resource tree, the attributes of their sibling nodes in the dynamic resource tree can be obtained as their contextual information for comparison. We meticulously examine attribute differences among child components under the same parent component, such as *size* or other characteristics to predict the attributes of newly added components.
- *UI Design issues (considering developer's intent).* Considering the developer's design intent requires combining the different features contained in the static resource tree and dynamic resource tree. As developers may define each group of components in the layout file in advance, the static resource tree built from compiled resource files will contain clearer group division, facilitating precise identification of intentional UI design. The dynamic resource tree is used to determine the status of each group of components when runtime. For example, in Figure 3 (UI Design of Navigation), the navigation bar can be found to contain a set of Button components with the "selected" attribute value using the static resource tree, while the dynamic resource tree obtains the runtime "selected" attribute value to identify the state changes of this group of components. Likewise, when dealing with *EditText* components, analyzing attribute values helps distinguish between hint-text and display text, resulting in more accurate accessibility asses.

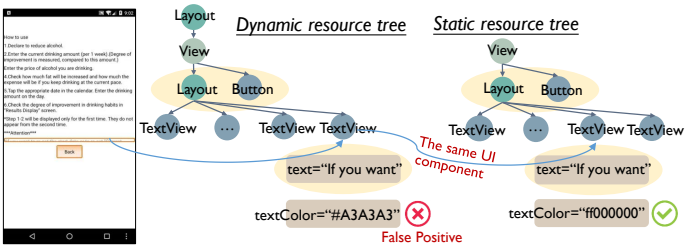

Fig. 5: A real example of using resource trees to obtain the contextual information of the detected UI component for accessibility detection.

Fig. 5 is an example of using resource trees to obtain the contextual information of the detected UI component for accessibility detection. When using ATF to detect the attributes of a *TextView* component that violates accessibility rules and

Algorithm 2: Context-aware Issue Detection

Input: $\mathcal{RF}s$: Decompiled resource files for static tree generation.
$\mathcal{RL}s$: Runtime layouts for dynamic tree generation.
$View$: The View of detected UI scenario captured by SUPERVISOR.

Output: $\mathcal{FP}s$: Issues incorrectly identified by ATF.
$\mathcal{TI}s$: Accessibility issues identified in the UI scenario.

```
1  sTree ← GenerateStaticTree(RFs)
2  dTree ← GenerateDynamicTree(RLs)
3  foreach comp ∈ View do
4      warning ← CheckedByATF(comp)
5      if IsExternalOrIsInvalid(comp, warning) then
6          FPs.append(comp, warning)
7      if InStaticAndInDynamic(comp, sTree, dTree)
       then
8          FP,TI ← CompareAndCheck(comp, sTree,
           dTree)
9      else
10         FP,TI ← CompareWithSibling(comp, sTree,
           dTree)
11     FPs.append(FP)
12     TIs.append(TI)
13 return FPs, TIs
```

poses a low contrast issue, we first locate the *TextView* in the *dynamic resource tree* and obtain an attribute that uniquely identifies the component, such as *text="if you want"*. Then we use this attribute to locate the position of the *TextView* in the static resource tree. To confirm the correctness of the positioning, the hierarchical structure of the *TextView* and its parent nodes (Layout and Button marked by yellow shadow in Fig. 5) also needs to be consistent in both resource trees. By comparing the *TextView* in two resource trees, we found that the *textColor="#A3A3A3"* reported by ATF is inconsistent with the pre-defined attribute of *textColor="ff000000"* in the static resource tree, indicating the existence of a false positive due to folded component and other reasons.

To check the accessibility of the detected UI components by our newly-defined checking rules combining the ATF and the contextual information gained from resource trees, we developed a monitoring app called SUPERVISOR, which integrates our newly-defined checking rules. SUPERVISOR can detect accessibility issues on the currently displayed UI page during the UI scenario exploration of the tested app. SUPERVISOR runs in the background as a service and accesses the accessibility features on Android devices through the Accessibility API provided by ATF. The Accessibility API scans the UI page of the app to access various components and checks if they are accessible by using our rules.

Algorithm 2 elaborates on context-aware issue detection aimed at utilizing newly-defined checking rules for accessibility issue detection. The input includes decompiled resource files $\mathcal{RF}s$, runtime layouts $\mathcal{RL}s$, and the View $View$ of detected UI scenario captured by SUPERVISOR, with the output distinguishing between false positives $\mathcal{FP}s$ caused by ATF and true issues $\mathcal{TI}s$. The procedure entails constructing static and dynamic resource trees $sTree, dTree$ using decompiled resource files and runtime layouts, respectively (Lines 1∼2). For each component $comp$ in the $View$, we first check if this $comp$ violates ATF accessibility rules, and remark warning $warning$ if true. (Lines 3∼4) For each the warning $warning$ detected by ATF, we determine whether the warning stems from external dependencies or invalid components by comparing attributes within the resource trees (Lines 5∼6). If a component exists in both resource trees (Line 7), we evaluate the consistency between the trees to decide if the warning should be classified as a false positive FP or a true issue TI (Line 8). In other instances, we compare sibling node attributes, which is especially common among dynamically constructed components (Lines 9∼10). The algorithm ultimately returns a collection of both categories of issues (Lines 11∼13).

A11yScan uses ApkTool to disassemble APK resource files. To collect UI information, we used UIAutomator [66] and ADB [67] to dump the layout hierarchy files and capture the screenshots when running an app on an Android Emulator based on Android 9.0 (Google APIs). The emulator's screen resolution was 1080px x 1920px, aligning with the specifications outlined in the WCAG [40], [41].

IV. Experiments

To evaluate the effectiveness of A11yScan, we propose the following three research questions:

- **RQ1: (Evaluation of Overall Performance)** What is the overall performance of A11yScan compared with the state-of-the-art accessibility testing tool?
- **RQ2: (Evaluation of Scenario-driven Exploration)** What is the performance of scenario-driven exploration compared with state-of-the-art UI exploration and GUI testing tools?
- **RQ3: (Evaluation of Context-aware Detection)** What is the performance of context-aware detection for reducing false positives of accessibility issues?

Dateset-1: We randomly selected 100 apps from F-Droid [68] and Google Play [69], including 50 open-source apps and 50 closed-source apps of various types, such as social media and shopping. This diversity was intentionally included to ensure a broad representation of UI designs and user interactions.

Dataset-2: We obtained an available dataset of 20 apps that were used as a part of their experiments from GPTDroid [31], which was released from Themis benchmark [70].

A. RQ1: Evaluation of Overall Performance

1) Setup: To evaluate the effectiveness of A11yScan, we compared the detection results of *Dataset-1* with Xbot [7], a state-of-the-art automated accessibility issue detection tool with a higher activity coverage rate, utilizing Google Accessibility Scanner [17]. We used Xbot and A11yScan respectively to detect the number of accessibility issues in these apps while recording the number of UI scenarios detected and the average activity coverage. By comparing these results, we evaluated the effectiveness of A11yScan in addressing false negatives of existing accessibility testing tools.

TABLE II: Comparison results for effectiveness evaluation of Issue Detection between A11yScan and Xbot.

	Avg. act. cov.	# UI scenarios	# True Issues (# Reduced FPs)
Xbot	68%	460	2,321 (471)
A11yScan	75%	952	3,991 (1,070)

TABLE III: Comparison results for effectiveness evaluation of UI exploration between A11yScan and SceneDroid.

	Avg. act. cov.	# UI scenarios
SceneDroid	66%	661
A11yScan	75%	952

Furthermore, we recorded the execution time of the 100 apps in *Dataset-1* and calculated the average time to show A11yScan's time performance, which helps us better understand how A11yScan performs in practical use.

2) Result: The results for the effectiveness evaluation are shown in Table II. The column "Avg. act. cov." represents average activity coverage while the column "# UI scenarios" refers to the number of unique UI scenarios in the 100 apps explored by these two tools. It is worth noting that for scrollable UI pages, A11yScan scrolls and captures screenshots until the page becomes non-scrollable to detect accessibility issues. Subsequently, A11yScan stitches these screenshots together, considering all scrolling screenshots of the same UI page as one new UI scenario. The column "# True Issues (# Reduced FPs)" represents the total number of issues and false positives in those issues detected by each tool in detected UI scenarios. Since Xbot does not have the ability to identify false positives, we manually went through all detected issues and filtered the false positives accordingly. The difference is that the number of false positives was counted automatically for A11yScan, with the aid of its context-aware detection module.

Table II shows the performance of Xbot and A11yScan in detecting accessibility issues. On the detected accessibility issues, Xbot identified 2,321 true issues in 460 UI scenarios, while A11yScan identified 3,991 true issues in 952 UI scenarios. There were 471 and 1,071 false positives respectively that have been removed from the number of true issues in their detection results. We conclude that A11yScan can detect 1.7X ($\frac{3,991}{2,321}$) more accessibility issues than Xbot further validates A11yScan's superiority in accurately detecting more comprehensive issues by increasing the coverage of UI scenarios. The experimental results present that A11yScan can effectively mitigate false negatives resulting from insufficient exploration. Specifically, the more detected issues can reduce the false negative rate by 41.84% ($\frac{3,991-2,321}{3,991}$). Exploring more accessibility issues, accompanied by more UI scenarios, highlights the importance of improving UI scenario coverage in detecting accessibility issues, which also indicates that the scenario-driven UI exploration method has contributed to reducing false negatives by exploring more UI scenarios. Through extensive UI scenario coverage, A11yScan captures the different attributes of UI components across different UI scenarios, thereby detecting more potential accessibility issues by identifying how UI components behave and present themselves in diverse UI states. Meanwhile, more comprehensive UI exploration also brings higher activity coverage, with 0.68 of Xbot and 0.75 of A11yScan. And this advantage in activity coverage is also attributed to the broader exploration of UI scenarios.

In addition to evaluating the effectiveness of A11yScan in UI scenario exploration and issue detection, we also recorded the execution time of A11yScan. According to our statistical results, A11yScan takes an average of 15.15 minutes to execute one app, while Xbot takes 2.52 minutes with simpler UI exploration.

Answer to RQ1: Compared with the state-of-the-art accessibility testing tool Xbot, A11yScan performs well in both UI scenarios (460 vs. 952) and issue detection (2,321 vs. 3,991), with nearly twice as many UI scenarios and 1.7X more issues, reducing the false negative rate by 41.84%.

B. RQ2: Evaluation of Scenario-driven Exploration

1) Setup: To evaluate A11yScan's ability to explore UI scenarios based on the scenario-driven exploration method, we compared A11yScan with SceneDroid [38], a state-of-the-art UI exploration tool, which is better than current existing app testing tools in obtaining UI scenarios with the highest activity coverage based on the experimental results in their paper. We deployed SceneDroid to perform UI exploration on 100 apps in *Dataset-1* and record the number of UI scenarios explored and the average activity coverage.

Besides, we also compared A11yScan with a recent state-of-the-art GUI testing tool GPTDroid [31], which has been proven to be superior to other GUI testing tools [29], [30], [32], [71], [72]. Because GPTDroid is not open-sourced and its tool and used dataset are not available, we used a released dataset (i.e., *Dataset-2*), which is a part of their experimental dataset (20/86) and compared them with their shared experimental results in terms of activity coverage in their pre-print version. We highlight that the average number of activities used in *Dataset-2* is 12, which is higher than their all experimental dataset (12 vs. 9), therefore, the average Activity coverage of these 20 apps should not differ significantly from the overall average value (i.e., 71%) of their experimental apps.

2) Result: Table III shows the performance of SceneDroid and A11yScan in exploring UI scenarios. The column "Avg. act. cov." represents average activity coverage while the column "# UI scenarios" refers to the number of unique UI scenarios in the 100 apps explored by these two tools. On the number of UI scenarios, SceneDroid and A11yScan explored 661 and 952 unique UI scenarios with activity coverage at 0.66 and 0.75, respectively. Remarkably, A11yScan outperforms the state-of-the-art UI exploration tool SceneDroid, which highlights the effectiveness of the scenario-driven UI exploration method adopted by A11yScan in achieving a more extensive exploration of UI scenarios. Although SceneDroid focuses on state changes triggered by interactions with UI components on each UI scenario, such as drawers, it only focuses on UI new scenarios directly generated by one interaction. The advantage of A11yScan lies in considering the indirect changes to a particular UI scenario caused by multiple interactions. As shown in Fig. 1, user interaction flows of adding a

new category in *CategoriesActivity* can lead to the update of *MainActivity*. There are usually transition relationships between different activities in an app, and one Activity can be launched through interaction with another Activity. Although A11yScan can improve activity coverage by interacting with other UI scenarios, it may be limited by factors such as data and environment dependencies in some cases. This means that even if efforts are made to increase the coverage of UI scenarios, it may not be possible to fully cover all activities, as some activities may depend on specific data or environmental conditions that may be difficult to simulate or replicate.

The average activity coverage of A11yScan on *Dataset-2* is 73%, and GPTDroid is 71% mentioned in their paper [31]. A11yScan achieved a comparable result on activity coverage with GPTDroid. While the LLM-enhanced approach would have more advantages in launching activities on filling more valuable inputs in *EditText*, thereby improving the activity coverage, A11yScan also uses a more comprehensive scenario-driven UI exploration strategy to explore more activities and scenarios, which can complement each other in UI exploration. In terms of required effort, A11yScan achieves high activity and UI scenario coverage with minimal effort and high automation by combining static and dynamic analysis. In contrast, LLM-based GUI testing tools often require substantial computational resources for inference and may still need manual intervention to fine-tune model behavior and handle specific app contexts, leading to increased complexity and effort. Due to the unavailability of GPTDroid, the ability of UI exploration cannot be further compared in our experiments.

Answer to RQ2: Compared with the leading UI exploration tool SceneDroid, A11yScan can explore more UI scenarios (661 vs. 952), which is attributed to the novel scenario-driven UI exploration. Additionally, when compared with the recent state-of-the-art GUI testing tool utilizing the cutting-edge technique LLM, A11yScan maintains comparable activity coverage, highlighting its effectiveness.

C. RQ3: Evaluation of Context-aware Detection

1) Setup: To evaluate A11yScan's ability to reduce false positives based on the context-aware detection method, we investigated how many issues can be filtered by such a method. We also investigated the different types of detected false positive issues and their corresponding characteristics.

To validate the accuracy of A11yScan in identifying false positives, we conducted a user study. We recruited three participants familiar with app development from our university and provided them with training to acquire professional knowledge and identification skills for accessibility issues. For example, using WCAG2.1 [41] as a unified guideline to explain accessibility standards in detail to participants. Then, we provided data from A11yScan's results on a randomly selected 20 apps with a total of 963 detected issues to them, which included UI screenshots that highlighted marked accessibility issues. In these screenshots, potential true issues and false positives were explicitly marked. Then, the first two participants independently marked the issues detected

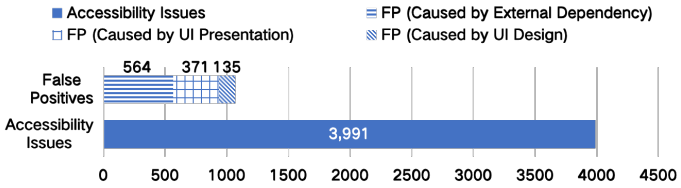

Fig. 6: Issue proportion after runtime context-aware analysis.

by A11yScan for correctness, distinguishing between actual issues and false positives. Subsequently, the third participant reviewed the assessments of the first two participants, focusing on areas of inconsistency in the markings and making the final judgment. Finally, we documented and analyzed the results and feedback from all participants to understand A11yScan's performance in identifying and categorizing accessibility issues, particularly in handling false positives.

2) Result: During the phase of the context-aware detection method, A11yScan constructs and uses resource trees containing rich contextual information to assist in identifying accessibility issues and false positives, and the results are shown in Fig. 6. "FP (Caused by External Dependency)" represents External Dependency and Invalid issues recognized by A11yScan, "FP (Caused by UI Presentation)" represents UI Presentation issues, and "FP (Caused by UI Design)" represents UI Design issues. These types of issues are all false positives that are not necessarily problematic in the actual use of the app. From the experimental results, the number of false positives caused by these three reasons introduced above is 564, 371, and 135, respectively. Their total accounts for 21% of all detected issues, which demonstrates the necessity and effectiveness of using context-aware analysis in A11yScan.

In these false positives, External Dependency issues, as well as UI Presentation issues, arise from the lack of contextual information, leading to incorrect identification by detection tools, which are not actual accessibility issues of the app and are meaningless to developers. Therefore, these issues are not reported in the issue reports. Note that, despite appearing to violate accessibility standards, our investigation reveals that most users and app developers view special UI design issues (135/1,070 FPs) as improvements to the overall user experience rather than accessibility issues, making them contentious regarding their accessibility impact and therefore classified as false positives, requiring special consideration. By offering precise issue reports that identify and annotate these cases, we assist developers in balancing user experience enhancement with app accessibility. Since these design decisions are intentional, the final decision on whether to fix these issues is left to the developers.

After manual verification by three participants, they identified 154 true false positives among 963 issues in 20 apps, while A11yScan detected 170 false positives, resulting in an accuracy rate of 90.56%. The causes of the remaining 16 misjudgments are as follows. *(1)* The attributes of some components dynamically change in the code, resulting in inconsistency with the pre-defined attributes obtained from decompiled resource files. *(2)* Some dynamically added com-

ponents may not have a parent or sibling node, which makes it difficult to obtain valuable contextual information on attributes and hierarchy during accessibility detection. This resulted in discrepancies between the resource trees of the source file obtained by decompiling and the resource tree of the current page components, leading to erroneous judgments. Additionally, we found that the construction of resource trees facilitates rapid and accurate manual verification of issues. For instance, the contextual information about components provided by resource trees helps to verify whether components belong to External Dependency. These findings demonstrate the vital role of resource trees in providing contextual information in A11yScan.

Answer to RQ3: A11yScan can help reduce 21% false positive rate on our collected dataset caused by three types of reasons, which is attributed to the novel context-aware detection method. The false positive detection accuracy is 90.56% verified by a user study.

V. Discussion

A. Limitations

The limitations of A11yScan are as follows: (1) Although A11yScan has increased the coverage of UI scenarios, we still face limitations such as data dependency and environment dependency in further improving activity coverage. The initiation of certain activities may depend on specific data or environmental conditions, which are difficult to simulate or replicate. (2) In addition, using randomly generated inputs to fill *EditText* components performs average in complex scenarios. Although the result of Activity coverage is comparable with the GPTDroid shown in Section IV-B, for the ability to fill up for specific types of input requirements, cutting-edge techniques like the large language model would somehow help by combining the traditional methods. Therefore, we will attempt to incorporate the LLM-enhanced approach in the future to further enhance our UI exploration capabilities. (3) Furthermore, A11yScan uses the newly-defined checking rule for accessibility detection, and it may overlap some accessibility issues that only arise when using assistive services (such as screen readers). Therefore, in future research, a combination of both rules and assistive technology should be considered to obtain the best detection results.

B. Threats to Validity

While our selected apps of experimental subjects in Section IV cover a diverse range of representative open-source and closed-source apps, certain specific types of apps may not have been fully considered, perhaps with unique characteristics, which limits our assessment of A11yScan's generalization ability in different app types.

C. Cross-Platform Application

Applying A11yScan to platforms beyond Android is feasible and meaningful, as accessibility standards and the severity of issues are unified across platforms. Although the design frameworks vary across different platforms, the core methodology and technology of A11yScan can be applied to other platforms through targeted adjustments and optimizations to detect accessibility issues. However, challenges may arise during implementation, as the tool is specifically designed for Android applications and relies on Android-specific frameworks and APIs, necessitating effort for specific adjustments.

VI. Related Work

A. Accessibility Testing

Accessibility testing is crucial for ensuring that Android apps are usable by all users, including those with disabilities. As we thoroughly introduced in Section II, there are a large number of tools that have been proposed to test the accessibility of apps, including static analysis tools [13], [14], [16] and dynamic analysis tools (i.e., manual dynamic method [17], [19], [20], [53], [54], [73], script-based dynamic method [21]–[23], [55]–[57], and automated dynamic method [7], [24], [58]–[60]). Compared with these existing accessibility testing tools, owing to the new proposed techniques (i.e., scenario-driven UI exploration and context-aware issue detection) in this paper, A11yScan performs significantly better than the current state-of-the-art methods (details in Section IV-A).

B. UI Exploration

Automated UI exploration tools are an important component of mobile app testing. Many automated UI exploration tools have been proposed and used in Android. Many automated UI exploration tools have been researched such as ATG [74], WTG [75], STG [76], and Storyboards [36], [37], [77], and SceneDroid [38]. They built different graphs with different granularity to help explore the UI pages of Android apps. Among them, although Storyboard has achieved good results in UI page exploration, it tends to overlook the importance of interactive components on the page and is slightly lacking in the ability to obtain new activity-dependent UI scenarios. SceneDroid demonstrates its advantage in extracting activity-dependent UI scenarios but falls short in addressing dynamically generated UI states resulting from user interactions. Overall, although there are various Android automated page exploration tools available, they are limited to the coverage of UI scenarios. Compared with them, A11yScan achieves a significantly higher coverage of UI scenarios (details in Section IV-B) by leveraging two key strategies in the technique of scenario-driven UI exploration (details in Section III-A).

VII. Conclusion

In this paper, we proposed A11yScan to automatically test and detect accessiblity issues for Android apps. Compared with the existing tools, A11yScan achieves significantly better performance in both UI scenario exploration and issue detection, which reduces the hidden danger of false negatives. Additionally, A11yScan introduces a context-aware issue detection to reduce false positives. The experimental results demonstrate its effectiveness and efficiency.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China (Grant No. 62472309).

REFERENCES

[1] N. Hassan, S. O. Gillani, and M. S. Nawaz, "Mobile applications for people with disabilities: Barriers and opportunities," *Journal of Accessibility and Design for All*, vol. 10, no. 2, pp. 120–140, 2020.

[2] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock, "An epidemiology-inspired large-scale analysis of Android app accessibility," *ACM Transactions on Accessible Computing (TACCESS)*, vol. 13, no. 1, pp. 1–36, 2020.

[3] 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)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1323–1334. [Online]. Available: https://doi.org/10.1145/3377811.3380392

[4] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock, "Examining image-based button labeling for accessibility in Android apps through large-scale analysis," in *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, 2018, pp. 119–130.

[5] W. H. Organization *et al.*, *World Report on Disability 2011*. World Health Organization, 2011.

[6] 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.

[7] 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, 2021.

[8] 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 *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1255–1267. [Online]. Available: https://doi.org/10.1145/3611643.3616329

[9] S. Chen, Y. Zhang, L. Fan, J. Li, and Y. Liu, "Ausera: Automated security vulnerability detection for android apps," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[10] S. Yan and P. Ramachandran, "The current status of accessibility in mobile apps," *ACM Transactions on Accessible Computing (TACCESS)*, vol. 12, no. 1, p. 3, 2019.

[11] C. Silva, M. M. Eler, and G. Fraser, "A survey on the tool support for the automatic evaluation of mobile accessibility," in *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, 2018, pp. 286–293.

[12] J. Zhu, K. Li, S. Chen, L. Fan, X. Xie *et al.*, "A comprehensive study on static application security testing (sast) tools for android," *IEEE Transactions on Software Engineering*, 2024.

[13] PMD, "PMD," https://pmd.github.io, 2023, accessed on 27 April 2023.

[14] Checkstyle, "Checkstyle," https://checkstyle.sourceforge.io, 2023, accessed on 27 April 2023.

[15] FindBugs, "FindBugs," http://findbugs.sourceforge.net, 2023, accessed on 27 April 2023.

[16] Google, "Lint," 2021. [Online]. Available: https://developer.android.com/studio/write/lint

[17] ——, "Accessibility Scanner," 2021. [Online]. Available: https://developer.android.com/studio/intro/accessibility/scanner

[18] Microsoft, "Accessibility Insights," 2023. [Online]. Available: https://accessibilityinsights.io/

[19] IBM Corporation, "IBM Accessibility Checklist," https://www.ibm.com/able/guidelines/ci162/accessibility-checklist.html, 2023, accessed: April 26, 2023.

[20] N. Salehnamadi, Z. He, and S. Malek, "Assistive-technology aided manual accessibility testing in mobile apps, powered by record-and-replay," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–20.

[21] N. Patil, D. Bhole, and P. Shete, "Enhanced UI Automator Viewer with improved Android accessibility evaluation features," in *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT)*. IEEE, 2016, pp. 977–983.

[22] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, "Espresso vs. eyeautomate: An experiment for the comparison of two generations of android gui testing," in *Proceedings of the Evaluation and Assessment on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 13–22. [Online]. Available: https://doi.org/10.1145/3319008.3319022

[23] 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 *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–11.

[24] N. Salehnamadi, F. Mehralian, and S. Malek, "Groundhog: An automated accessibility crawler for mobile apps," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[25] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[26] 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*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 245–256. [Online]. Available: https://doi.org/10.1145/3106237.3106298

[27] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 269–280.

[28] 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*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 469–480. [Online]. Available: https://doi.org/10.1145/3377811.3380382

[29] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1070–1073.

[30] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 153–164. [Online]. Available: https://doi.org/10.1145/3395363.3397354

[31] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," *arXiv preprint arXiv:2310.15780*, 2023.

[32] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile gui testing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1355–1367.

[33] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu, "Are mobile banking apps secure? what can be improved?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 797–802.

[34] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global android banking apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1310–1322.

[35] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 115–127.

[36] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 596–607.

[37] S. Chen, L. Fan, C. Chen, and Y. Liu, "Automatically distilling storyboard with rich features for android apps," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 667–683, 2022.

[38] X. Zhang, L. Fan, S. Chen, Y. Su, and B. Li, "Scene-driven exploration and gui modeling for android apps," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1251–1262.

[39] Google, "Accessibility Testing Framework," 2021. [Online]. Available: https://github.com/google/Accessibility-Testing-Framework

[40] W. W. W. C. (W3C), "Web Content Accessibility Guidelines (WCAG) 2.0," 2008. [Online]. Available: https://www.w3.org/TR/WCAG20/

[41] ——, "Web Content Accessibility Guidelines (WCAG) 2.1," 2018. [Online]. Available: https://www.w3.org/TR/WCAG21/

[42] G. Inc., "Google Accessibility," 2021. [Online]. Available: https://developer.android.com/guide/topics/ui/accessibility

[43] I. O. for Standardization (ISO), *Ergonomics of human-system interaction - Part 210: Human-centred design for interactive systems*. ISO, 2019. [Online]. Available: https://www.iso.org/standard/77520.html

[44] U. S. A. Board, "Revised 508 standards - united states access board," 2017. [Online]. Available: https://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-ict-refresh/final-rule/text-of-the-standards-and-guidelines

[45] H. Swan, B. G. F. Williams, B. J. Avilla *et al.*, "Draft bbc mobile accessibility standards and guidelines," *BBC internet blog*, 2013.

[46] G. D. Service, "Accessibility Guidance - Government Digital Service," 2021. [Online]. Available: https://www.gov.uk/guidance/accessibility-requirements-for-public-sector-websites-and-apps

[47] Google, "Android Accessibility Developer Guidelines," https://developer.android.com/guide/topics/ui/accessibility/, 2023, accessed: April 26, 2023.

[48] A. Inc., "Apple Accessibility Developer Guidelines," https://developer.apple.com/accessibility/, 2023, accessed: September 30, 2023.

[49] U.S. Department of Justice, "Americans with disabilities act," https://www.ada.gov/, 1990, accessed: 26-April-2023.

[50] Federal Communications Commission, "Communications and video accessibility act," https://www.fcc.gov/general/communications-and-video-accessibility-act-cvaa, 2010, [Accessed: 26-April-2023].

[51] U. Nations, "Convention on the rights of persons with disabilities," 2006. [Online]. Available: https://www.un.org/development/desa/disabilities/convention-on-the-rights-of-persons-with-disabilities.html

[52] E. Union, "Communication from the commission to the european parliament, the council, the european economic and social committee, and the committee of the regions: European accessibility act - accessible europe: European accessibility act," 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:52016DC0628&from=EN

[53] A. S. Alotaibi, P. T. Chiou, and W. G. Halfond, "Automated detection of talkback interactive accessibility failures in android applications," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 232–243.

[54] L. Burkhart, "Stepping stones to switch access," *Perspectives of the ASHA Special Interest Groups*, vol. 3, no. 12, pp. 33–44, 2018.

[55] R. C. Grüner, M. Dziadzka, J. Lerch, and V. Daburon, "Robotium: An android test automation framework," in *Proceedings of the 6th International Conference on Mobile Technology, Applications, and Systems*. Association for Computing Machinery, 2009, pp. 1–8.

[56] I. Pivotal Software, "Robolectric: Unit testing android in the jvm," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services Companion*. Association for Computing Machinery, 2012, pp. 317–318.

[57] Appium. (2023) Appium. Appium Community. Accessed on 26th April 2023. [Online]. Available: http://appium.io

[58] 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.

[59] P. Xu, L. Li, Z. Liang, G. Wang, W. Li, T. Li, and J. Chen, "PUMA: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 473–488.

[60] ForApp, "forApp Mobile Accessibility Inspection Solution," http://www.forapp.org, April 2018.

[61] M. C. Burnett, J. D. Hollan, and S. R. M. Epsilonlroy, "Monkey: a tool for exploring the accessibility of software user interfaces," in *Proceedings of the 2005 ACM SIGCHI International Conference on Human-Computer Interaction*. ACM, 2005, pp. 421–430. [Online]. Available: https://doi.org/10.1145/1054972.1055033

[62] Google, "Monkeyrunner," 2021. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner

[63] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of Android applications by constructing activity launching contexts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 457–468. [Online]. Available: https://doi.org/10.1145/3377811.3380347

[64] Y. Li, J. Ouyang, B. Mao, K. Ma, and S. Guo, "Data flow analysis on android platform with fragment lifecycle modeling and callbacks," *EAI Endorsed Transactions on Security and Safety*, vol. 4, no. 11, 12 2017.

[65] R. Rivest, "The md5 message-digest algorithm," Tech. Rep., 1992.

[66] Google, "UI Automator," 2021. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[67] Google Developers, "Android debug bridge (adb)," https://developer.android.com/studio/command-line/adb, 2023, accessed: 2023-11-18.

[68] F-Droid. (2022) F-Droid. [Online]. Available: https://f-droid.org

[69] G. Play. (2022) Google Play Store. [Online]. Available: https://play.google.com

[70] 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*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 119–130. [Online]. Available: https://doi.org/10.1145/3468264.3468620

[71] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 23–26.

[72] 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*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 481–492. [Online]. Available: https://doi.org/10.1145/3377811.3380402

[73] B. Leporini, M. C. Buzzi, and M. Buzzi, "Interacting with mobile devices via voiceover: usability and accessibility issues," in *Proceedings of the 24th Australian computer-human interaction conference*, 2012, pp. 339–348.

[74] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 641–660. [Online]. Available: https://doi.org/10.1145/2509136.2509549

[75] S. Yang, H. Wu, H. Zhang *et al.*, "Static window transition graphs for Android," *Automated Software Engineering*, vol. 25, pp. 833–873, 2018. [Online]. Available: https://doi.org/10.1007/s10515-018-0237-6

[76] D. Lai and J. Rubin, "Goal-driven exploration for Android applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 115–127.

[77] Y. Zhang, S. Chen, and L. Fan, "A web-based tool for using storyboard of android apps," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2023, pp. 117–121.