2018 IEEE 11th International Conference on Software Testing, Verification and Validation

Automated Accessibility Testing of Mobile Apps

Marcelo Medeiros Eler*, Jose Miguel Rojas§, Yan Ge†, and Gordon Fraser‡
*University of Sao Paulo (USP), Brazil
§University of Leicester, UK
†University of Sheffield, UK
‡University of Passau, Germany

E-mail: marceloeler@usp.br, j.rojas@leicester.ac.uk, yge5@sheffield.ac.uk, gordon.fraser@uni-passau.de

***Abstract*—It is important to make mobile apps accessible, so as not to exclude users with common disabilities such as blindness, low vision, or color blindness. Even when developers are aware of these accessibility needs, the lack of tool support makes the development and assessment of accessible apps challenging. Some accessibility properties can be checked statically, but user interface widgets are often created dynamically and are not amenable to static checking. Some accessibility checking frameworks analyze accessibility properties at runtime, but have to rely on existing thorough test suites. In this paper, we introduce the idea of using automated test generation to explore the accessibility of mobile apps. We present the MATE tool (*Mobile Accessibility Testing*), which automatically explores apps while applying different checks for accessibility issues related to visual impairment. For each issue, MATE generates a detailed report that supports the developer in fixing the issue. Experiments on a sample of 73 apps demonstrate that MATE detects more basic accessibility problems than static analysis, and many additional types of accessibility problems that cannot be detected statically at all. Comparison with existing accessibility testing frameworks demonstrates that the independence of an existing test suite leads to the identification of many more accessibility problems. Even when enabling Android's assistive features like contrast enhancement, MATE can still find many accessibility issues.**

I. Introduction

An estimated 15% of the world population experience some form of disability and face barriers in regular life [41], including access to information and communication technology (ICT). The ICT infrastructure is increasingly shifting towards applications for mobile devices ("apps"), where accessibility thus is becoming a more pressing concern. For example, Fig. 1 shows screenshots of the Google Play app, used on Android devices to access the official app store. While this is a well designed and pleasant user interface, users with visual impairment may struggle with the low contrast ratio of the category labels at the top of the left hand screen. Moreover, users with physical and motor skill issues might struggle to interact with the highlighted buttons since they are too small.

Assistive technologies such as screen readers can support such users in many situations, but these also require the developers to correctly implement accessibility features of the apps. For example, the highlighted buttons in the Google Play app (Figure 1) all have the same content description (e.g., "Mark comment as helpful" for all buttons with the thumbs up image, and "Options" for all buttons with three dots). Users relying on screen readers would not be able to distinguish

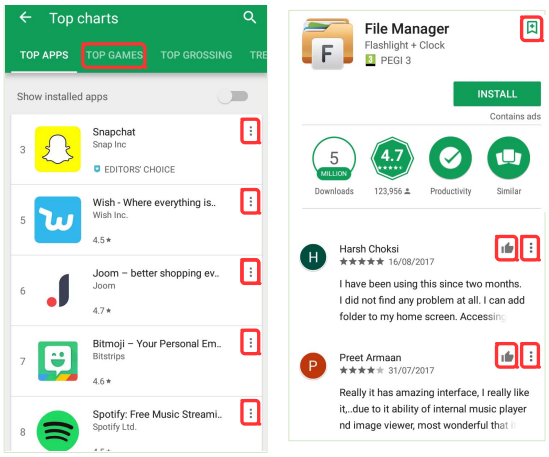


Fig. 1: Two activities of the Android Google Play app: The top labels have low contrast and may be difficult to see; the highlighted buttons all have the same content description, making them unusable for users with screen readers.

which comment a "Mark comment as helpful" button is related to, and thus would not be able to use the app properly.

Since accessibility is a recognized problem (e.g., [13], [21], [23]), and relevant international organizations such as the W3C have proposed standards and guidelines to ensure accessibility, there are tools that support developers. For example, static analysis tools like Android Lint [19] can check whether user interface components have labels that can be used by screen readers. However, the dynamic nature of Android apps means that only a subset of the user interface can be statically checked, and more advanced properties cannot be checked statically at all since they depend on the target device and configuration. Developers can manually check individual activities of Android apps using the Accessibility Scanner [21], and testing frameworks such as Espresso [17] and Robolectric [18] can check accessibility properties at runtime while system test cases are executed. However, developers are unlikely to write dedicated accessibility tests, and existing test suites tend to be weak [29].

To address these problems, we propose to use automatic test generation to find accessibility flaws in mobile apps.

0-7695-6388-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICST.2018.00021

Automated test generation and accessibility testing are perfectly complementary techniques: Automated test generation explores the behavior of an app under test, and thus overcomes the dependency of accessibility checking frameworks such as Espresso or Robolectric on existing tests; by checking properties at runtime, it also overcomes the limitations of static analysis tools like Android Lint. Automated test generation uses accessibility standards and guidelines as test oracles, and supports developers in improving the accessibility of their apps.

In detail, the contributions of this paper are:

- We introduce the notion of automated mobile accessibility testing (Section III), where an app is automatically explored for accessibility problems.
- We introduce the MATE tool (Mobile Accessibility TEsting), which implements automated accessibility testing (Section III-B). The initial prototype of MATE provides checks for several types of visual impairment and motor skill issues, and implements efficiency optimizations tailored towards the use for accessibility testing.
- We empirically evaluate automated accessibility testing by (1) measuring the effects of MATE's accessibility-related efficiency improvements, (2) comparing the flaws found by MATE with those found by Android Lint, Espresso and Robolectric tests, and (3) determining whether MATE can find many flaws even if the assistive features of modern mobile operating systems are enabled (Section IV).

In our experiments on open source mobile apps, our MATE prototype was able to detect more accessibility violations related to missing content descriptions than the Android Lint static checker, and many other types of accessibility problems that cannot be detected statically. Comparison against Android accessibility testing frameworks such as Espresso and Robolectric revealed that MATE can detect vastly more accessibility violations since it does not rely on an existing test suite. While modern mobile operating systems offer assistive features such as contrast enhancement, our experiments show that MATE can find many flaws even when these features are activated. These encouraging results reveal the potential offered by automated accessibility testing, and open up new possibilities for future work on improving accessibility of mobile apps. Ultimately, accessibility could become an automatically checked requirement for inclusion in an app store.

II. Background

The use of mobile devices has increased drastically in recent years, and research in testing mobile apps has made substantial progress (e.g., [12], [37]). Unlike other non-functional properties like security and privacy (e.g., [15]), performance (e.g., [32]), and energy-efficiency (e.g., [8]), and in spite of its criticality for users with disabilities, accessibility remains a relatively neglected property in mobile software testing. This is evidenced by accessibility being mentioned only tangentially or not at all in recent surveys in the field [3], [38], [43].

To some extent, the lack of research on accessibility testing can be attributed to the lack of precise definitions of what accessibility means in the context of mobile apps. Google provides some guidelines on how to make Android apps more accessible [23]. Although following these guidelines is encouraged, they are often ignored by developers. In fact, the use of these accessibility guidelines is so rare that it has become a necessity to develop specific applications exclusively for disabled users (e.g., [1]). However, the World Wide Web Consortium (W3C)—the main international standards organization for the World Wide Web—is currently working towards adapting their web accessibility guidelines and providing separate, specific guidelines for mobile accessibility [13]. In this paper, we adopt their definition for mobile accessibility: *"Mobile accessibility" refers to making websites and applications more accessible to people with disabilities when they are using mobile phones and other devices.*

A. Manual Mobile Accessibility Testing

Manually testing the accessibility of mobile apps largely involves exploring and inspecting the apps, and checking each encountered user interface component. For example, Android app developers can use the UI Automator Viewer [25] tool provided by Google to inspect the layout hierarchy and UI components of a visible screen and to manually check those properties of the UI components that they believe to be relevant for accessibility. If they encounter problematic property values, they can improve the app accordingly.

Accessibility Scanner [21] provides suggestions to improve the accessibility of an app. It checks four properties: content labels (item label missing, duplicate item labels, input fields with content descriptions, and labels with redundant information), clickable items (clickable subsets of text items and duplicate clickable bounds), contrast, and touch area size. These accessibility checks are provided by the Google's Accessibility Test Framework [16], a library that collects several accessibility-related checks on Android apps. The drawback of this approach, however, is that the developer must activate the tool on the device in each screen of the app to get the results. This means it requires manual exploration of the application, which might not scale for larger apps or frequent testing.

An alternative lies in exploring the apps from the perspective of users requiring system-provided assistive features that aim to improve accessibility of apps, such as Google's TalkBack and Switch Access [24]. TalkBack adds spoken, audible, and vibration feedback to the device, while Switch Access lets the user interact with devices using switches instead of the touch screen. Certain types of accessibility flaws, such as missing text descriptions, are easily detected using this approach. However, the obvious disadvantage of this approach is that it can be time-consuming, costly (e.g., Switch Access requires dedicated hardware), and labor-intensive. For iOS apps, there are also utilities intended to help developers interact with their apps from the perspective of users with disabilities [4]. In the same spirit, several accessibility-checking toolkits have been produced in more academic contexts [14], although they are not as far-reaching as the ones mentioned before.

Research on accessibility testing has mostly addressed web applications, and assumes manual exploration. Mankoff et

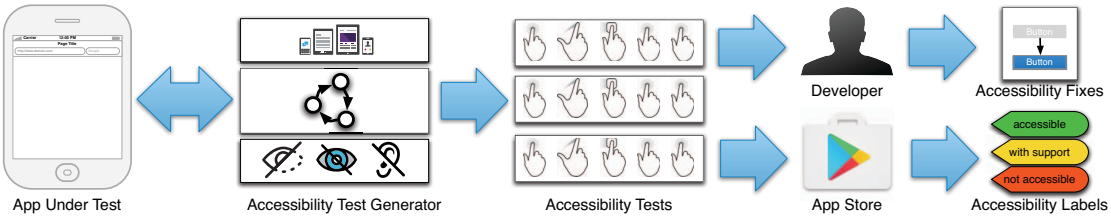


Fig. 2: Overview of automated accessibility testing: An automated test generator, parameterized with device configurations and accessibility properties, interacts with the app under test. Any accessibility flaws found are exported as tests to support developers in fixing the flaws, and which can be used by app stores to classify or tag the apps based on their accessibility status.

al. [36] compared different manual approaches to finding accessibility problems in web applications. Brajnik [10] suggested a more systematic approach than standard reviews (i.e., conformance testing) to assess web accessibility. Malý et al. [35] proposed a model-based approach to assess accessibility without the need to involve target users in the testing process. More specific to mobile accessibility, Billi et al. [9] proposed a unified methodology for early manual assessment of accessibility and usability of mobile applications. Using Brajnik's methodology, Yesilada et al. [42] studied the accessibility barriers experienced by disabled users and mobile users in general. While many of these principles also apply to mobile apps, in general manual approaches are laborious, resulting in insufficient testing of the accessibility of mobile apps.

B. Automated Tools and Approaches

Developers can resort to tools and frameworks to partially automate accessibility testing tasks. Android Lint [19] statically checks all files of an Android project, and reports all structural problems found. One category of these problems are accessibility issues, and Lint specifically checks whether non-textual visual components have a content description or input fields are properly labeled, and whether the implemented custom views properly override all methods that are intended to provide assistive technologies with accessibility-related information. Lint is limited to check only static properties that come from the source code, and cannot check dynamic properties such as size and contrast, for instance. Another limitation is that it can only check properties of components that are created during development time. However, in practice, many Android apps create content and components dynamically at runtime.

An alternative to static analysis is offered by testing frameworks such as Espresso [17] and Robolectric [18], which include features to check accessibility properties dynamically, during test execution. Similar tools exist for iOS, too: EarlGrey [20] is known as the Espresso for iOS, and KIF [28] is an open-source iOS integration test framework with accessibility-checking capabilities (comparable to Robolectric).

Espresso targets GUI testing and allows testers to simulate user interactions with visual components. In each test class, the developer can turn on accessibility checkers to automatically raise exceptions when any component activated by a test fails to comply with the accessibility properties defined by the W3C [13]. Robolectric is a unit testing framework, which can also test user interaction. While Espresso tests are executed using an emulator of the Android device, Robolectric tests are executed directly on the Java virtual machine, and directly invoke the app's API. This makes test execution faster compared to frameworks where tests are executed on an actual device or emulator. Developers can also enable accessibility checks in test cases, but unlike Espresso, accessibility checks must be explicitly declared for each component.

Both Espresso and Robolectric use the Google Testing Framework [16], and therefore implement the same accessibility checks as Accessibility Scanner. Even though they implement a comprehensive number of accessibility checks, their effectiveness highly depends on the thoroughness of the test cases available for the app under test. Developing dedicated accessibility test cases for all accessibility properties might not scale, considering the number of widgets and properties that typically need to be checked. Hence, even though developers may be aware of the accessibility features they should implement, there might be no test cases to reveal accessibility problems in the application.

In order to increase automation, the AMC tool uses model checking to evaluate various usability properties of vehicular apps, some of which are related to accessibility (e.g., number of words on a screen) [30]. Such properties can, in principle, also be checked during automated testing. For example, PUMA is a generic framework that can be customized for various types of dynamic analyses, such as checks for button sizes [26]. In general, many different automated GUI testing methods and tools have been proposed for mobile applications recently [2], [6], [7], [11], [12], [33], [34], [37], [39]. Most of these tools focus on interacting with the application under test with the aim of finding defects related to exceptional behavior and crashes. In this paper, we investigate the alternative of using dynamic analysis techniques to automatically check accessibility requirements.

III. Automated Accessibility Testing

This section describes the idea of automating accessibility testing, as well as the MATE tool, which implements this approach for Android apps.

A. General Principle

Fig. 2 illustrates the proposed approach: At the core of the approach is a test generator, which interacts with an app under test. The test generator is configured with different disabilities

it should check for as well as different types of devices that should be supported. The test generator explores the activities of the application under test and identifies accessibility flaws. For each accessibility flaw detected during the exploration, an accessibility test case is generated which fails as long as the accessibility flaw persists in the app. These tests can serve two purposes: (i) as guidance for developers to improve the app's accessibility; and (ii) as quality metric for app stores to label publicly available apps, for example, as *accessible* if no failing accessibility test exists for the app; *with support* if the operating system's assistive features (e.g., contrast enhancement) are sufficient to overcome all accessibility flaws, or *not accessible* if the app exhibits accessibility flaws (i.e., failing accessibility tests) even when assistive technologies are used.

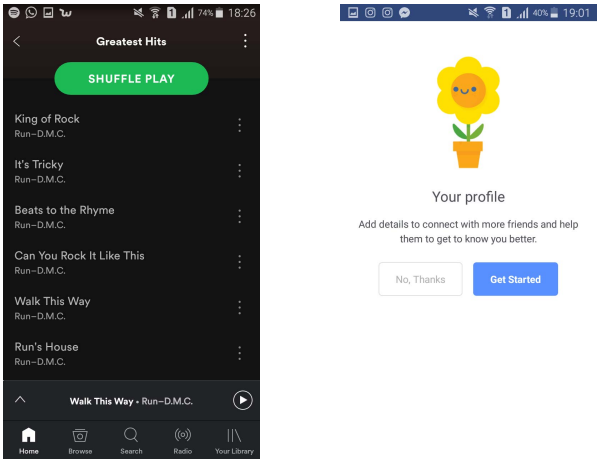
B. MATE: The Mobile Accessibility Testing Tool

Mobile Accessibility Testing (MATE) implements the approach described in Fig. 2 for Android applications. Structurally, an Android app is composed of activities, services, content providers and broadcast receivers. An *activity* is the entry point of interaction of an app with the user, i.e., a graphical user interface screen (see Fig. 1). An activity works as a container of components called *widgets*, e.g., buttons, text fields, toggle switches, etc., which allow the user to interact with the app and navigate through its activities. One single activity may present different widget configurations (i.e., different arrangement and states of the widgets) as the user interacts with the app.

MATE implements a random strategy to explore the app under test. Rather than producing random events on random coordinates, MATE identifies all possible user interactions in the current screen and randomly selects one of them. The exploration stops when a time budget is met. In order to assure that the exploration can come back to the initial states, the app is restarted after a given number of events. When input data is required, it is generated at random: Text input is sampled from a dictionary of English words, and numbers are generated randomly. One of the goals of this exploration strategy is to reach as many different states as possible. Therefore, MATE detects which type of input is required in each input field to avoid raising exceptions due to incompatible data types, which might prevent certain states from being reached.

During exploration, MATE applies accessibility checks on the widgets encountered. Since some properties depend on the specific device the app is executed on, MATE can be re-executed several times on a set of target device configurations (e.g., different screen sizes). The approach is configurable in the accessibility properties that should be checked. At the end, MATE reports the number of unique flaws of each type (check) implemented, as well as a detailed report on all accessibility flaws found. To avoid redundant warnings, MATE reports the unique accessibility flaws by activity. Since the number of such flaws can be substantial for complex apps, we propose that the actionable output of an automated accessibility test generator is a set of specific tests for accessibility flaws.

MATE runs in the Android environment (emulators or physical devices) and uses the UIAutomator framework to interact with the app. It collects information on the activity on execution by means of accessibility information available on each view. MATE does not require the source code of the app under test. To use MATE, one simply needs to start the app under test on the device or emulator, and then execute MATE, which will then test that app until the stopping criterion is met.



(a) Spotify: The back button (〈) has missing speakable text, and the too small "⋮" buttons all have the same duplicate text.

(b) Facebook: The "No, Thanks" button has low contrast.

Fig. 3: Examples of accessibility flaws checked by MATE.

C. Accessibility Properties

General accessibility guidelines propose a great number of accessibility features which applications should implement in order to be suitable for users with most types of impairment. For the first version of MATE, we have selected the following accessibility properties that—without empirical evidence available—we deem most likely to jeopardize the user experience when operating and navigating through the application: (1) Speakable text, (2) touch size area, (3) contrast ratio, (4) duplicate clickable bounds, and (5) clickable span.

1) Speakable text: The screen reader function (e.g., Google's TalkBack app [22] or Apple's VoiceOver iOS app [5]) can be very helpful for users with visual impairment. It allows them to interact with their apps through synthesized, audible descriptions of the app's components. However valuable, the screen reader function will not be effective if the app's components are not properly labeled with informative textual descriptions. The guidelines specify how to set these labels in practice: There should not be more than one visual component with the same label so the user does not get confused. Input fields should provide a property hint or label instead of a content description, to prevent the screen reader from reading both, the content and the content description altogether.

Three checks in MATE are related to this property:

- *Missing speakable text* (see Fig. 3a): Non-textual widgets (e.g., image buttons) must provide alternative text descriptions to be used by assistive screen readers. Speakable text is also recommended for input fields.

- *Duplicate speakable text* (see Fig. 3a): If more than one non-textual widget has the same speakable text, a user using a screen reader may not know to which widget that action refers to. For instance, if several widgets have the speakable text "remove", how would the user know what he/she will remove by clicking that widget?
- *Editable content description*: Input fields should have their speakable text set as "hints" or "label by", not "content description". If the content description property is set, the screen reader will read it even when the input field is not empty, which could confuse the user who might not know what part is the text in the input field and which part is the content description.

Instead of using static analysis to run these checks, MATE uses the Android API class `AccessibilityNodeInfo` to retrieve, during runtime, properties of the widgets presented on the screen, including "content description" and "hint". Therefore, MATE can detect issues related to speakable text even when components are created or updated during runtime.

2) Contrast ratio: Mobile devices are likely to be used in different environments including outdoors, where sun glare can reduce the visibility of a mobile screen. Moreover, mobile app developers tend to combine multiple colors to highlight components, or to try to present information more effectively. In this context, using adequate contrast ratios can be crucial to render the application usable for users with visual impairment. The guidelines specify minimum contrast ratios for large (3:1) and small (4.5:1) texts. Consequently, MATE checks that the contrast ratio between the foreground and the background is at least 3 for larger visual components and 4.5 for small ones. For example, the "No, Thanks" button in the Facebook app (Fig. 3b) has a contrast ratio of only 1.35.

To calculate the contrast ratio, MATE takes a screenshot of the current screen, and extracts an image of the target widget with three pixels of surrounding border. A particular challenge hereby is that, even though at design level usually only one color is set for the background or for the foreground of a visual component, the rendered widgets may use shaded colors. Another challenge lies in non-standard visual components or images with different colors. To handle these cases, MATE uses the Otsu threshold formula [40] to differentiate colors by their likelihood of being in the background or in the foreground. After calculating the Otsu threshold color of the widget image, MATE calculates the predominant color in the background (i.e., colors below the threshold) and the predominant color in the foreground (i.e., colors above the threshold). The luminance for each color is calculated as well as the contrast ratio between the brightest color and the darkest color.

3) Touch size area: Since users with motor or visual impairment can struggle to identify and interact with small touchable components in mobile apps, the accessibility guidelines require that touchable areas must have a dimension of at least 48x48dp (density-independent pixels). Density-independent Pixels is an abstract unit that is based on the physical density of the screen (dpi). On the Android operating system, one dp is equivalent to one physical pixel on a 160 dpi screen. Therefore, the ratio of dp-to-pixel changes as the screen density changes. For example, one dp is equivalent to two physical pixels on a 320 dpi screen and three pixels on a 480 dpi. So, the ratio of dp-to-pixel is given by the formula: $densityRatio = (density/160)$.

Considering that the visual bounds of a widget is given by the pixel coordinates of the upper left point (x1,y1) and of the lower right point (x2,y2), MATE calculates the dimension of each widget in dp by using the following formulas: $targetHeight = (y2 - y1)/densityRatio$ and $targetWidth = (x2 - x1)/densityRatio$. Both the coordinates and the screen density are dynamically retrieved using the Android API, which means this property can be checked even when widgets are resized during runtime.

4) Duplicate clickable bounds: Two actionable widgets should not share the same clickable area. This usually happens when containers are set as clickable even when they contain clickable widgets. MATE can check this property by retrieving the coordinates of each widget on the screen during runtime using the Android API.

5) Clickable span: Developers can set words or expressions within a text to be clickable by declaring them as a "clickable span". However, clickable spans are inaccessible because individual spans cannot be selected independently in a single `TextView`, and because accessibility services are unable to call its method `onClick`. Users that rely on screen readers are the most affected by this flaw. If a clickable span is required, developers should use the `URLSpan` instead. This property is checked by identifying each clickable span of a text view and checking whether they are instances of `URLSpan`.

Like other tools related to accessibility checking, MATE also uses the Google's Accessibility Testing Framework (ATF) [16]. In particular, three checks were directly reused from ATF: missing speakable text, editable content description and clickable span. Two checks were adapted: Duplicate clickable bounds and touch size area did not differentiate between clickable and non-clickable widgets originally, and thus we included a condition in which only clickable widgets are checked. Furthermore, we changed the way clickable elements were identified, since ATF only checks widgets whose property "clickable" is set to true. However, there are cases in which that criterion omits clickable elements, thus limiting the analysis and exploration of the app. For instance, there are many cases in which a container is declared as clickable, but does not implement any click behavior, while their children are truly clickable but are declared as non-clickable. MATE identifies such situations in order to perform a more comprehensive analysis and to expand the exploration possibilities. The duplicate speakable text and contrast ratio checks are not based on ATF. In ATF, those checks are based on information extracted from the Android API class `View` instead of `AccessibilityNodeInfo`, like all other checks.

D. Optimizing Accessiblity Checks with State Abstraction

During app exploration, the same widget can be encountered many times. Checking the same accessibility properties on the same widgets over and over again potentially wastes time

and can lead to spurious warnings. In particular, accessibility checks can be computationally expensive since they may require image processing tasks, e.g., to calculate the contrast ratio of each visual component. On the other hand, checking each widget only once may miss accessibility issues since the same widget can have different properties depending on the program context. To increase efficiency, MATE therefore uses a model-based strategy (as is generally common in app testing [12]), where a model of the GUI is produced automatically during the application exploration. The model represents the screen configurations (states) and the transitions between screens; each transition is caused by a system event or by user interaction, such as a click or a typed text, for instance.

During exploration, MATE checks whether a new screen state has been discovered after each interaction and then updates the GUI model accordingly. Accessibility checks are only run when a new screen state has been discovered. Therefore, the criteria to distinguish screen states are important. A coarse grained criterion, for example, would consider that a new screen state has been reached only when a new activity is executed. A fine grained approach would consider any change within the widget properties as a new state. Considering accessibility testing, the chosen criterion should allow the test generation to test the same widget in different states since it might affect its accessibility properties. An unchecked item in a form, for example, can change color when it is checked, which may change its contrast ratio with the background.

MATE uses four out of the five criteria defined by Baek et al. [7] to distinguish when a new state has been reached. The first and second criterion compares the Java package name and activity name, respectively. The third criterion compares the hierarchy of the widget compositions, and the fourth criterion compares basic properties of each widget. A check box may be, for example, checked or unchecked, and since the state may have an influence on the accessibility of the widget it is important that we include the widget states in our state abstraction. The fifth criterion defined by Baek et al. [7] compares the content of the widgets, so that an input text field with different content would lead to different screen states. MATE does not use this criterion since it might easily cause state explosion, while not being directly relevant to the accessibility properties of the widgets.

The model underlying MATE's exploration can be represented as a directed graph in which each node represents a screen state and each screen represents an event. An event consists of a widget and a specific action, such as clicking a button. In Android apps, activities may have dynamic and complex user interfaces that can map to different screen configurations, i.e., different widgets compositions. Therefore, one activity can map to different screens at runtime.

IV. Evaluation

To evaluate the viability of automated accessibility testing, we empirically address the following research questions.

First we look at the effectiveness of the state abstraction implemented in MATE (Section III-D):

TABLE I: Experimental setup. All apps were downloaded from the F-Droid repository of open-source Android apps [31].

RQ	MATE vs. ...	#Apps	Selection	#Activities		
				Min.	Avg.	Max.
1	MATE (no state abs.)	10	Random	3	12.6	35
2	Lint	50	Random	1	7.6	50
3	Espresso	12	Systematic	2	8.0	24
4	Robolectric	13	Systematic	3	12.0	31
5	High Contrast Feature	50	Random	1	7.6	50

RQ1. How effective is MATE's state abstraction?

Then, we compare MATE to existing mobile testing frameworks with accessibility testing capabilities:

RQ2. How does MATE compare to static analysis with Android Lint?

RQ3. How does MATE compare to the accessibility testing frameworks relying on existing tests?

Finally, we investigate whether the accessibility flaws MATE detects can be avoided by enabling Android's assistive features:

RQ4. Can MATE find accessibility flaws that cannot be avoided by enabling Android's assistive features?

A. Experimental setup

Although MATE does not require the source code of the apps it is applied to, some of the tools we compare it to do. Therefore, we selected all apps under test from the F-Droid repository [31] of open source apps. The selection of apps differs between the RQs, and is explained in detail below. In all cases, we excluded (a) apps which we did not succeed to build and (b) apps that can only be run in specific countries or which control specific physical devices or sensors. During the selection process, we excluded only one app. All experiments were executed on a laptop with Intel Core i7-4510U, CPU 2.00GHz quad-core, 16 GB of memory, running Ubuntu 16.04.2 LTS. Experiments with MATE were executed in four emulators running in parallel. All emulators used images set up with Android Marshmallow (version 6.0, API level 23). More details on the setup of each RQ are presented in Table I and described below. Some of the checks in MATE are configurable; in particular, we used the W3C recommendations of a minimum dimension of 48x48dp for touchable components, and a minimum acceptable contrast ratio for regular size components of 4.5.

1) RQ1: In order to evaluate whether our choice of state abstraction achieves a performance improvement and does not cause too many false negatives, we compared MATE in two configurations: With checks applied only when new abstract states are detected, and checks applied on all widgets on all screens. We randomly selected 10 mobile apps from F-Droid, and ran MATE on them for a fixed number of 1,500 events, with 5 repetitions to account for randomness. The difference in the number of unique flaws found estimates the loss in precision caused by the state abstraction. Then, we ran MATE on the same apps for a fixed time budget of 30 minutes, and compared the number of unique flaws again to measure whether the performance gains pay off.

2) RQ2: Android Lint is the state of the art static analysis tool for Android apps and includes some accessibility checks. To compare MATE with Lint, we applied both tools on 50 apps from F-Droid. Since in many projects Lint can only be applied as part of the app's build process (using Gradle [27]), we selected only apps for which we managed to compile the source code. We used Lint v25.3.1 and Gradle v3.4. We applied MATE directly on the APK files (Android's installation package format) of the apps, using a time budget of 30 minutes per app, and 10 repetitions to account for randomness.

3) RQ3: The Espresso framework supports accessibility checks that developers must enable in their tests. To compare MATE to Espresso, we selected all F-Droid projects which contained tests using the Espresso framework. To locate these, we searched for specific import statements in the source code of the test classes. Out of 21 resulting projects, we excluded three whose test cases we could not run due to build issues we could not fix, and six projects in which a dependency to the Espresso framework was declared but no Espresso test case was found; the final selection thus contained 12 projects. Since none of these projects were explicitly configured to use Espresso's accessibility checks, we programmatically instrumented all test classes to enable these checks. As a result, during test execution, all widgets on the screen under test have their properties checked against the accessibility standards adopted by Espresso. We applied MATE on the same 12 apps for 30 minutes, with 10 repetitions.

The Robolectric framework performs accessibility checks using existing test cases. We again identified all projects in F-Droid referencing Robolectric (19), out of which 13 projects actually had tests. To collect the flaws detected by Robolectric, we instrumented all selected projects to enable the accessibility checks for each test case. Then, we executed all Robolectric test cases. To collect the accessibility flaws detected by MATE, we tested the same 13 apps for 30 minutes, with 10 repetitions.

4) RQ4: Many mobile devices provide users with features intended to accessibility issues. Considering the properties checked by MATE, the main relevant assistive feature provided by Android is the High Contrast Fonts feature, which adjust the color and outline the fonts to increase the contrast with the background. To check whether MATE can still find accessibility flaws even when this feature is enabled, we compared the performance of MATE with and without the High Contrast Fonts feature. For this experiment, we used the same 50 apps selected for the comparison against Lint. All apps were tested during 30 minutes, with 10 repetitions.

B. Threats to validity

1) App selection: We selected apps from the F-Droid repository because Lint, Espresso and Robolectric require source code. While we are not aware of any fundamental differences between F-Droid apps and others, we are unable to make strong generalizability claims. Different results may be observed, in particular, when comparing against Espresso and Robolectric for apps with stronger existing test suites.

2) Execution time: In our experiments, each instance of MATE was run for 30 minutes. While this suffices to draw conclusions about the tool's performance, studying the practicality of using this time budget in realistic industrial contexts is left for future work.

3) Emulation: All apps were executed in emulators with the same configuration (Android SDK 6 and API level 23). Testing apps in real devices and/or different configurations may change flaw detection rates, both by MATE and by the rest of tools. We preferred the emulation and single configuration setup because: i) it avoids a potentially high computational overhead; and ii) it suffices to show with statistical significance the effects of our automated testing approach.

4) Contrast: The contrast ratio calculated for each widget checked during app exploration are based on the Otsu thresholding method [40], which is used to distinguish colors by their likelihood of being in the background or foreground of an image. This technique may not be precise in some cases, leading MATE to produce false positives or true negatives. We countered this threat by carefully tuning and testing our implementation of the Otsu method.

5) User validation: The goal of this paper is to demonstrate the effectiveness of MATE at detecting accessibility flaws. By encoding the standard accessibility guidelines as automated oracles, we can be sure the flaws reported by MATE are genuine. For specific users, however, the guidelines may be overly strict, and so the flaws detected by MATE might not resonate with all end users (e.g., a button too small by the guidelines may be perfectly accessible by the intended users of the app). Validating the impact of accessibility flaws on end users will be essential, but falls out of the scope of this paper.

6) Accessibility flaws: Our evaluation comprises accessibility checks currently implemented in MATE or in tools under comparison. There are accessibility flaws that are not very amenable for automated detection. For instance, it is easy to check whether a content description or hint property is empty or not, but it is not easy to check whether their values are appropriate and meaningful for end users. In general, MATE, and automated testing overall, is not yet able to validate accessibility scenarios in which semantics plays an important role. Further research will be needed in this direction.

C. RQ1: How effective is MATE's state abstraction?

In order to determine how many accessibility flaws are missed because of the state abstraction, we executed MATE for a fixed number of events (1,500). Table II (columns 2-7) lists the average number of checks performed, time spent, and number of flaws found. These results show that using the Abstraction configuration drastically reduces the number of accessibility checks performed by MATE (column 3 vs. column 2), therefore making the tool more efficient (time comparison in columns 4-5). In terms of effectiveness, the number of flaws found by MATE using the Abstraction configuration is slightly lower (7.5 flaws per app on average). This indicates a loss of precision when the state abstraction criterion does not correctly recognize a new state.

Table II ("Fixed Time Budget" columns) also shows the results of evaluating MATE's abstraction using a fixed time budget. This represents a more practical application scenario for an automated testing tool (e.g., in continuous integration). These results justify the use of the state abstraction as MATE's default behavior: the performance improvement of state abstraction outweighs the loss of precision and leads to a substantial average increase of 19.58 detected accessibility flaws.

RQ1: Given a fixed time budget, state abstraction increases the number of accessibility flaws MATE *can find.*

D. RQ2: How does MATE *compare to static analysis with Android Lint?*

Table III compares the number of accessibility flaws found by Android Lint and MATE, overall and by type of flaw. Overall, MATE detects significantly more flaws than Lint does. This is mainly because Android Lint can only find accessibility issues that can be statically detected. Specifically, it checks for flaws related to missing speakable text for screen readers (Table III, "Missing Text"), and cases in which custom views do not implement the event handling methods as required by accessibility services ("Clickable View").

Considering only checks performed by both tools ("Missing Text"), MATE found 5.7 flaws on average per app, while Lint found 4.5 on average. For 10 projects there is almost no difference (less than 1 flaw difference); for 5 projects both tools could not detect any flaws; for 28 projects MATE detected more flaws; and for 7 projects Lint detected more flaws. Across all apps in the experiment, MATE performs significantly better for this type of flaw, with an average effect size A_{12}=0.67.

RQ2: MATE *finds more missing text descriptions than Lint, and many additional types of accessibility flaws.*

One advantage of Lint over MATE is that it can perform its analysis very quickly and can cover all activities for which a layout has been statically defined. While MATE can capture flaws on dynamically created and combined layouts, it can only do that if it manages to reach the activities where these flaws occur. In this experiment, MATE covered an average of 80% of the activities of the apps under test. The activity coverage is usually lower when context data is required to access parts of the app, such as authentication, for example. Since Lint does not execute code, it does not depend on activity coverage.

There are also flaws that Lint can detect by analyzing the source code, which dynamic analysis cannot detect. For example, MATE cannot check if a custom view has implemented all event handlers properly ("Clickable View"), while Lint detected this type of flaw in 10 out of 50 projects. The overall number of flaws of this type considering all projects was 16, which may suggest these flaws are relatively rare in Android apps.

E. RQ3: How does MATE *compare to accessibility testing frameworks relying on existing tests?*

Table IV shows the results of the comparison between MATE and Espresso. The overall number of flaws detected by MATE was significantly higher than the number of flaws found by executing Espresso tests with accessibility checks enabled. This is largely due to MATE being able to detect a large number of size and contrast flaws. Espresso only detected flaws of type "Size" and "Missing Text". While there are two apps (OCReader and MicroPinner) on which both tools detected the same number of "Size" flaws, in the majority of apps MATE significantly outperformed Espresso for this flaw type. The same pattern is observed for "Missing Text" flaws, both tools performed equally for the OCReader app, but MATE was more effective (at different degrees) for the rest of apps.

Notice that these results are relative to the existing instrumented test cases on which Espresso relies to explore the different activities and states in the app under test. In our experiments, the activity coverage achieved by Espresso tests and MATE is quite similar. Intuitively, the more tests are available, the higher the chances for Espresso to detect more flaws due to increased activity coverage. However, if the existing test cases are intended to test only specific user interactions, many parts of the possible screens a user can access within a single activity will be neglected, and corresponding accessibility flaws will remain undetected. Moreover, this limitation imposes a higher cost for Android developers who want to use Espresso to assess the accessibility of their apps. MATE, on the other hand, does not require existing test cases and automatically explores different activities, and the states therein, aiming at checking the highest number of visual components it can reach.

Table V shows the results of the comparison between MATE and Robolectric. Like Espresso, Robolectric depends on existing tests and we found only 12 projects with existing Robolectric tests. Executing Robolectric tests with accessibility checks enabled did not reveal any accessibility flaws. Our conjecture is that Robolectric tests are generally intended as unit tests rather than system tests emulating user inputs. As such, these test exercise only few user interactions and thus provide few opportunities for accessibility checks to reveal flaws. Furthermore, in a large number of instances, Robolectric tests resort to mocking Android activities instead of executing them for real. Therefore, even when some activities with accessibility issues are referred in the test case, no accessibility flaw is revealed since the target activity is not actually loaded. As Robolectric runs test cases on the JVM, not on an emulator or real device, we could not collect the number of activities executed during the process.

RQ3: MATE *finds more accessibility flaws than Robolectric and Espresso, which rely on existing test cases.*

F. RQ4: Can MATE *find accessibility flaws that cannot be avoided by enabling Android's assistive features?*

Fig. 4 shows boxplots comparing the average number of contrast issues detected by MATE in two different configurations: when the text contrast enhancement feature is disabled (Off) and when it is enabled (On). When this feature is enabled, the contrast ratio between the foreground and the background for each text element increases. The result of this

TABLE II: RQ1. Comparison of MATE's performance with and without state abstraction.

App	Fixed Number of Events						Fixed Time Budget			
	Checks		Time (seconds)		Flaws		Checks		Flaws	
	Full	Abst.	Full	Abst.	Full	Abst.	Full	Abst.	Full	Abst.
Book Catalogue	305,062.0	28,066.4	4,618.2	2,328.0	162.4	132.2	48,896.4	20,661.6	57.8	124.0
Daily Money	511,678.4	17,699.4	7,754.4	3,152.6	76.2	70.8	84,259.2	13,845.8	39.6	63.8
KeepScore	375,854.6	19,521.6	5,175.8	2,824.8	103.8	95.2	69,932.6	13,724.0	66.2	72.4
LibreOffice Viewer	242,158.6	8,196.0	2,998.8	1,899.6	59.0	49.4	85,313.8	7,238.4	32.8	38.4
My Expenses	396,202.8	46,390.8	4,206.0	2,420.4	136.2	111.0	61,938.8	36,215.4	83.2	135.2
OI Shopping List	222,675.8	40,117.8	4,589.8	3,333.6	68.2	60.4	63,658.0	31,688.6	51.2	67.4
OpenTasks	553,765.2	47,373.6	5,500.4	2,266.2	148.6	107.2	217,876.4	43,041.8	115.0	125.6
Pomodoro Tasks	117,150.6	15,672.8	3,092.8	2,349.8	52.6	47.6	41,927.0	11,598.2	46.2	47.6
Repay	174,864.8	10,451.2	2,951.2	2,007.0	49.8	58.6	63,204.6	9,407.4	30.8	38.6
RoomMates	234,408.2	19,579.0	3,692.8	2,470.2	43.2	32.2	78,800.6	9,400.4	39.2	44.8

TABLE III: RQ2. Comparison of number of flaws found by MATE and Lint.

App	Activity Coverage		Overall		Size		Contrast		Duplicate Text		Missing Text		Editable Content		Click Bounds		Click Span		Clickable View	
	Lint	MATE	Lint	MATE	Lint	MATE	Lint	MATE	Lint	MATE	Lint	MATE	Lint	MATE	Lint	MATE	Lint	MATE	Lint	MATE
arXiv mobile	-	0.8	5.0	30.8	-	9.8	-	14.4	-	0.0	5.0	4.3	-	0.0	-	2.3	-	0.0	0.0	-
BipolAlarm	-	1.0	0.0	5.0	-	3.4	-	1.6	-	0.0	0.0	0.0	-	0.0	-	0.0	-	0.0	0.0	-
Birthday Calendar	-	0.8	0.0	32.9	-	8.8	-	18.6	-	0.0	0.0	3.0	-	0.0	-	2.5	-	0.0	0.0	-
BMI Calculator	-	1.0	0.0	4.0	-	0.0	-	2.0	-	0.0	0.0	2.0	-	0.0	-	0.0	-	0.0	0.0	-
Book Catalogue	-	0.7	33.0	136.3	-	63.3	-	33.8	-	14.8	31.0	21.6	-	0.0	-	2.8	-	0.0	2.0	-
Budget	-	0.9	0.0	51.5	-	20.2	-	18.9	-	10.4	0.0	0.8	-	0.0	-	1.2	-	0.0	0.0	-
Budget Watch	-	0.8	1.0	91.5	-	21.7	-	58.3	-	1.0	1.0	6.4	-	0.0	-	4.1	-	0.0	0.0	-
Cat Avatar Generator	-	1.0	3.0	5.9	-	0.0	-	1.7	-	1.5	3.0	2.5	-	0.0	-	0.2	-	0.0	0.0	-
CIDR Calculator	-	1.0	1.0	29.1	-	16.0	-	2.4	-	3.4	0.0	4.1	-	0.0	-	3.2	-	0.0	1.0	-
Clock+	-	0.6	20.0	62.3	-	13.1	-	35.9	-	0.0	16.0	13.2	-	0.0	-	0.1	-	0.0	4.0	-
Concurseiro	-	1.0	0.0	10.4	-	5.0	-	3.4	-	0.0	0.0	1.0	-	0.0	-	1.0	-	0.0	0.0	-
CuprumPDF	-	1.0	0.0	5.0	-	0.5	-	4.0	-	0.0	0.0	0.5	-	0.0	-	0.0	-	0.0	0.0	-
Daily Money	-	0.7	24.0	59.2	-	16.7	-	19.3	-	0.0	24.0	23.1	-	0.0	-	0.1	-	0.0	0.0	-
DroidWeight	-	0.5	9.0	70.9	-	31.7	-	25.8	-	3.9	9.0	8.5	-	0.0	-	1.0	-	0.0	0.0	-
EH17 Schedule	-	0.8	14.0	80.1	-	19.8	-	56.4	-	1.1	12.0	1.0	-	0.0	-	1.8	-	0.0	2.0	-
ExprEval	-	0.5	0.0	15.8	-	7.8	-	2.1	-	2.0	0.0	3.8	-	0.0	-	0.1	-	0.0	0.0	-
Free Fall	-	0.9	7.0	13.6	-	2.5	-	8.9	-	0.0	7.0	2.2	-	0.0	-	0.0	-	0.0	0.0	-
Frozen Bubble	-	0.8	0.0	10.6	-	3.1	-	4.3	-	0.0	0.0	3.2	-	0.0	-	0.0	-	0.0	0.0	-
Heriswap	-	1.0	0.0	0.0	-	0.0	-	0.0	-	0.0	0.0	0.0	-	0.0	-	0.0	-	0.0	0.0	-
Ithaka Board Game	-	1.0	16.0	25.0	-	0.2	-	3.3	-	0.1	16.0	21.3	-	0.0	-	0.1	-	0.0	0.0	-
KeepScore	-	0.7	10.0	75.7	-	16.9	-	44.4	-	2.0	10.0	11.0	-	0.0	-	1.4	-	0.0	0.0	-
KISS launcher	-	0.7	3.0	49.3	-	17.4	-	26.0	-	2.6	2.0	2.4	-	0.0	-	0.9	-	0.0	1.0	-
LaiCare	-	0.7	0.0	26.2	-	9.2	-	10.8	-	0.0	0.0	6.2	-	0.0	-	0.0	-	0.0	0.0	-
Lexica	-	1.0	1.0	19.0	-	2.6	-	13.9	-	0.1	0.0	1.3	-	0.0	-	1.1	-	0.0	1.0	-
Metrodroid	-	0.5	3.0	9.9	-	4.1	-	4.8	-	0.0	3.0	0.0	-	0.0	-	1.0	-	0.0	0.0	-
mGerrit	-	0.8	5.0	107.8	-	34.1	-	61.6	-	8.6	4.0	2.6	-	0.0	-	0.9	-	0.0	1.0	-
Mileage	-	0.5	8.0	71.3	-	10.7	-	13.4	-	17.3	8.0	28.3	-	0.0	-	1.6	-	0.0	0.0	-
My Expenses	-	0.5	1.0	119.6	-	45.1	-	56.6	-	1.0	1.0	5.5	-	0.0	-	11.4	-	0.0	0.0	-
Notable Plus	-	0.9	12.0	70.4	-	15.5	-	36.9	-	1.6	12.0	16.0	-	0.0	-	0.3	-	0.0	0.0	-
Nounours and friends	-	0.6	0.0	4.9	-	2.4	-	2.5	-	0.0	0.0	0.0	-	0.0	-	0.0	-	0.0	0.0	-
OI Notepad	-	0.8	5.0	37.3	-	9.3	-	17.8	-	3.2	5.0	7.0	-	0.0	-	0.0	-	0.0	0.0	-
OI Shopping List	-	0.4	8.0	56.2	-	24.0	-	16.6	-	4.4	8.0	10.8	-	0.0	-	0.4	-	0.0	0.0	-
OpenSudoku	-	0.6	2.0	17.3	-	1.8	-	10.5	-	0.0	2.0	5.0	-	0.0	-	0.0	-	0.0	0.0	-
OpenTasks	-	0.7	16.0	137.2	-	35.2	-	75.7	-	6.0	14.0	15.3	-	0.0	-	5.0	-	0.0	2.0	-
packlist	-	0.8	0.0	42.2	-	8.8	-	24.2	-	2.4	0.0	4.2	-	0.0	-	1.5	-	0.0	0.0	-
PReVo	-	0.8	0.0	24.8	-	13.9	-	6.7	-	1.3	0.0	1.7	-	0.0	-	1.2	-	0.0	0.0	-
React	-	1.0	1.0	7.5	-	1.1	-	5.4	-	0.0	0.0	1.0	-	0.0	-	0.0	-	0.0	1.0	-
Repay	-	0.7	1.0	44.9	-	8.7	-	32.0	-	2.1	1.0	0.8	-	0.0	-	1.3	-	0.0	0.0	-
Retro Breaker	-	1.0	0.0	8.8	-	2.4	-	6.4	-	0.0	0.0	0.0	-	0.0	-	0.0	-	0.0	0.0	-
Rocket Guardian	-	1.0	0.0	1.3	-	0.2	-	0.0	-	0.1	0.0	1.0	-	0.0	-	0.0	-	0.0	0.0	-
RoomMates	-	0.7	18.0	39.6	-	13.8	-	14.4	-	4.0	18.0	7.5	-	0.0	-	0.0	-	0.0	0.0	-
ShellsMP	-	0.7	0.0	6.1	-	6.1	-	0.0	-	0.0	0.0	0.0	-	0.0	-	0.0	-	0.0	0.0	-
Shiurim	-	1.0	0.0	7.2	-	5.6	-	0.6	-	0.0	0.0	1.0	-	0.0	-	0.0	-	0.0	0.0	-
Shorty	-	0.7	0.0	23.2	-	14.1	-	1.5	-	6.2	0.0	0.4	-	0.0	-	1.0	-	0.0	0.0	-
Smoke Reducer	-	1.0	0.0	4.6	-	0.7	-	3.7	-	0.0	0.0	0.2	-	0.0	-	0.0	-	0.0	0.0	-
Survival Manual	-	0.7	0.0	24.8	-	9.6	-	11.1	-	1.0	0.0	1.6	-	0.0	-	1.5	-	0.0	0.0	-
TimeTable	-	0.7	0.0	10.5	-	3.2	-	6.3	-	0.0	0.0	1.0	-	0.0	-	0.0	-	0.0	0.0	-
Track Work Time	-	0.7	0.0	94.9	-	33.2	-	53.4	-	2.7	0.0	2.8	-	0.0	-	2.8	-	0.0	0.0	-
Transdroid	-	0.7	1.0	64.3	-	25.7	-	20.8	-	0.9	0.0	10.1	-	0.0	-	6.7	-	0.0	1.0	-
TripSit	-	0.9	14.0	32.4	-	7.1	-	6.6	-	1.2	14.0	15.4	-	0.0	-	1.2	-	0.0	0.0	-
Average	-	0.8	4.8	39.6	-	12.5	-	18.0	-	2.1	4.5	5.7	-	0.0	-	1.2	-	0.0	0.3	-
Average A_{12}		-		0.99		-		-		-		0.67		-		-		-		-

TABLE IV: RQ3. Comparison of number of flaws found by MATE and Espresso.

App	Activity Coverage		Overall		Size		Contrast		Duplicate Text		Missing Text		Editable Content		Click Bounds		Click Span		Clickable View	
	Esp.	MATE	Esp.	MATE	Esp.	MATE	Esp.	MATE	Esp.	MATE	Esp.	MATE	Esp.	MATE	Esp.	MATE	Esp.	MATE	Esp.	MATE
Equate	1.0	0.5	8.0	43.6	7.0	10.2	0.0	23.6	0.0	0.0	1.0	8.7	0.0	0.0	0.0	1.1	0.0	0.0	-	-
Home Assistant	0.5	0.5	3.0	9.6	1.0	2.5	0.0	2.4	0.0	0.0	2.0	2.7	0.0	0.0	0.0	2.0	0.0	0.0	-	-
Kolab Notes	0.1	0.7	2.0	114.6	1.0	30.4	0.0	52.9	0.0	9.7	1.0	17.7	0.0	0.0	0.0	3.9	0.0	0.0	-	-
Kontalk	0.2	0.2	3.0	55.6	3.0	23.0	0.0	26.1	0.0	3.0	0.0	1.0	0.0	0.0	0.0	2.5	0.0	0.0	-	-
MicroPinner	1.0	1.0	7.0	16.0	7.0	7.0	0.0	6.0	0.0	2.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	-	-
My Expenses	0.4	0.5	7.0	119.6	7.0	45.1	0.0	56.6	0.0	1.0	0.0	5.5	0.0	0.0	0.0	11.4	0.0	0.0	-	-
OCReader	0.4	0.1	6.0	9.9	3.0	3.0	0.0	3.9	0.0	0.0	3.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	-	-
OI Shopping List	0.4	0.4	3.0	53.7	2.0	23.5	0.0	16.0	0.0	4.0	1.0	9.9	0.0	0.0	0.0	0.3	0.0	0.0	-	-
Omni Notes FOSS	0.1	0.3	1.0	144.2	0.0	50.1	0.0	64.7	0.0	6.5	1.0	19.1	0.0	0.0	0.0	3.8	0.0	0.0	-	-
OneTimePad	0.3	0.3	0.0	28.3	0.0	7.6	0.0	16.1	0.0	2.0	0.0	2.6	0.0	0.0	0.0	0.0	0.0	0.0	-	-
Orgzly	1.0	0.5	2.0	86.0	1.0	23.6	0.0	47.3	0.0	7.2	1.0	6.5	0.0	0.0	0.0	1.4	0.0	0.0	-	-
Poet Assistant	1.0	0.6	9.0	99.7	9.0	37.1	0.0	55.0	0.0	2.6	0.0	3.9	0.0	0.0	0.0	1.1	0.0	0.0	-	-
Average	0.5	0.5	4.2	65.2	3.4	21.9	0.0	31.0	0.0	3.2	0.8	6.8	0.0	0.0	0.0	2.3	0.0	0.0	-	-
Average A_{12}		0.48		1.00		0.92		1.00		0.87		0.95		0.50		0.83		0.50		-

TABLE V: RQ4. Comparison of number of flaws found by MATE and Robolectric.

App	Activity Coverage		Overall		Size		Contrast		Duplicate Text		Missing Text		Editable Content		Click Bounds		Click Span		Clickable View	
	Rob.	MATE	Rob.	MATE	Rob.	MATE	Rob.	MATE	Rob.	MATE	Rob.	MATE	Rob.	MATE	Rob.	MATE	Rob.	MATE	Rob.	MATE
And Bible	-	0.5	0.0	54.6	0.0	25.4	0.0	25.4	0.0	0.7	0.0	2.0	0.0	0.0	0.0	1.1	0.0	0.0	-	-
Dziennik	-	0.6	0.0	8.2	0.0	3.3	0.0	2.5	0.0	1.2	0.0	1.1	0.0	0.0	0.0	0.1	0.0	0.0	-	-
K-9 Mail	-	0.5	0.0	130.8	0.0	54.2	0.0	52.5	0.0	11.8	0.0	9.1	0.0	0.8	0.0	2.2	0.0	0.0	-	-
MGit	-	0.8	0.0	70.5	0.0	29.8	0.0	26.5	0.0	8.0	0.0	4.7	0.0	0.0	0.0	1.5	0.0	0.0	-	-
Nounours and friends	-	0.7	0.0	6.2	0.0	2.8	0.0	3.1	0.0	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	-	-
nusic	-	0.6	0.0	14.1	0.0	5.0	0.0	6.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.9	0.0	0.0	-	-
OCReader	-	0.1	0.0	10.6	0.0	3.6	0.0	3.8	0.0	0.0	0.0	3.2	0.0	0.0	0.0	0.0	0.0	0.0	-	-
PinDroid	-	0.1	0.0	7.2	0.0	3.6	0.0	1.2	0.0	1.2	0.0	1.2	0.0	0.0	0.0	0.0	0.0	0.0	-	-
Tachiyomi	-	0.6	0.0	96.0	0.0	30.1	0.0	44.8	0.0	2.1	0.0	14.8	0.0	0.0	0.0	4.2	0.0	0.0	-	-
Tickmate	-	1.0	0.0	37.0	0.0	8.3	0.0	21.5	0.0	0.0	0.0	3.0	0.0	0.0	0.0	4.2	0.0	0.0	-	-
Traccar Client	-	1.0	0.0	65.5	0.0	14.0	0.0	41.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	8.5	0.0	0.0	-	-
WiFiAnalyzer	-	1.0	0.0	66.2	0.0	19.7	0.0	38.4	0.0	1.2	0.0	5.1	0.0	0.0	0.0	1.8	0.0	0.0	-	-
Wikipedia	-	0.3	0.0	105.4	0.0	42.7	0.0	41.4	0.0	1.6	0.0	11.0	0.0	0.0	0.0	6.8	0.0	1.7	-	-
Average	-	0.6	0.0	50.8	0.0	19.0	0.0	22.6	0.0	2.3	0.0	4.6	0.0	0.1	0.0	2.2	0.0	0.1	-	-
Average A_{12}		-		1.00		1.00		1.00		0.79		0.93		0.51		0.84		0.54		-

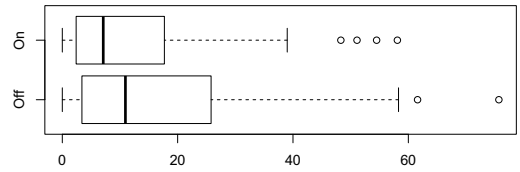


Fig. 4: RQ4. Comparison of number of contrast flaws found by MATE with and without the High Contrast Android feature.

comparison indicates that while this feature is indeed useful to fix many contrast flaws, it is not sufficient to fix them all. On average, the number of contrast flaws detected with the feature disabled is 18.00 (median 10.95), and when the feature is enabled, the average remains high: 13.50 (median 7.10). These results stress the importance of choosing suitable colors with contrast ratios that abide by the accessibility guidelines, but also demonstrates that MATE can be an effective addition to mobile app developers to assess how accessible their apps are even when assistive features are in use.

RQ4: MATE *finds many accessibility flaws that cannot be avoided with assistive features.*

V. Conclusions

An increasing shift towards mobile applications makes the problem of accessibility for disabled users an increasingly important concern. While many apps have well-designed and visually pleasing user interfaces, they are not accessible to users with common disabilities such as visual impairment. Accessibility is now not only a matter of desirable features anymore, but it is a mandatory requirement defined by law in many contexts. However, existing tools to support developers in making their apps accessible are either limited by the underlying static analysis, which may struggle with the dynamic nature of modern apps, or the dependence on existing test sets, which may not be of sufficient quality [29].

To address this problem, we introduced the concept of *automated accessibility testing*, which combines the benefits of automated test generation with the potential of dynamic analysis to discover accessibility problems. We have instantiated automated accessibility testing with the MATE prototype, a testing tool devised to automatically test accessibility properties. Initial experiments have shown that this approach overcomes the problems of all existing approaches: Accessibility properties are explored at runtime rather than statically, such that the dynamic properties of apps are taken into account. By integrating test generation, there is no dependence on an existing strong test suite. Our experiments show that MATE detects more accessibility problems than the static checker Android Lint, and more accessibility problems than the Espresso and Robolectric frameworks which rely on existing test sets.

The concept of automated accessibility testing opens up a wealth of research opportunities, for which our MATE prototype only serves as a first proof of concept. In particular:

- **User studies:** Our work to date is guided by general accessibility guidelines. However, an important aspect of evaluation would be to compare the issues discovered by tools with those encountered by actual users.
- **Developer studies:** We intend to evaluate how effective MATE is at helping real developers to identify and fix accessibility issues on their apps.
- **Accessibility properties:** Our MATE prototype implements some accessibility checks for visual, physical and motor skills impairment. However, there are other checks related to this and other impairments, such as hearing, for instance, that should also be considered.
- **Improved test generation:** While automated accessibility testing will benefit from any advances made in automated test generation, there is potential to optimize test generation specifically for accessibility testing. For example, accessibility properties could be integrated as optimization targets in search-based test generation techniques [37].
- **Accessibility certification:** As part of the overall methodology, we have outlined the idea of using automated accessibility testing as a means to automatically classify and certify apps that are accessible (Section III). This could be integrated directly into app stores where accessibility could be made a necessary requirement, or apps could be labeled based on their accessibility status. MATE could be used as an underlying tool executed in different configurations before publishing an app. We have taken an initial step towards this goal by creating https://android-mate.github.io, an open database of Android apps with accessibility flaws detected by MATE and which can be used by developers to improve their apps' accessibility support and by end users to know whether an app satisfies their accessibility needs.

REFERENCES

[1] D. Adams and S. Kurniawan. A blind-friendly photography application for smartphones. *SIGACCESS Access. Comput.*, pages 12–15, Jan. 2014.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of android applications. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 258–261, 2012.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins. Chapter 1 - testing android mobile applications: Challenges, strategies, and approaches. In *Advances in Computers*, volume 89, pages 1–52. Elsevier, 2013.

[4] Apple. Use Accessibility features on your iPhone, iPad, and iPod touch. https://support.apple.com/en-eg/HT204390, 2016.

[5] Apple. VoiceOver. http://www.apple.com/uk/accessibility/iphone/vision/, 2016.

[6] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *ACM SIGPLAN Notices*, 48(10):641–660, 2013.

[7] Y.-M. Baek and D.-H. Bae. Automated model-based android GUI testing using multi-level GUI comparison criteria. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 238–249. ACM, 2016.

[8] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 588–598, 2014.

[9] M. Billi, L. Burzagli, T. Catarci, G. Santucci, E. Bertini, F. Gabbanini, and E. Palchetti. A unified methodology for the evaluation of accessibility and usability of mobile applications. *Universal Access in the Information Society*, 9(4):337–356, 2010.

[10] G. Brajnik. Web accessibility testing: when the method is the culprit. In *Int. Conference on Computers for Handicapped Persons (ICCHP)*, pages 156–163. Springer, 2006.

[11] W. Choi, G. Necula, and K. Sen. Guided GUI testing of android apps with minimal restart and approximate learning. *ACM SIGPLAN Notices*, 48(10):623–640, 2013.

[12] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? (e). In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[13] W. W. W. Consortium. Web Accessibility Initiative (WAI). https://www.w3.org/WAI/mobile/. Last accessed February 2017.

[14] M. Gemou, J. B. Montalva Colomer, M. F. Cabrera-Umpierrez, S. de los Rios, M. T. Arredondo, and E. Bekiaris. Validation of toolkits for developing third-generation android accessible mobile applications. *Universal Access in the Information Society*, 15(1):101–127, 2016.

[15] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, editors, *Int. Conference on Trust and Trustworthy Computing (TRUST)*, pages 291–307. Springer Berlin Heidelberg, 2012.

[16] Google. Accessibility testing framework. https://github.com/google/Accessibility-Test-Framework-for-Android.

[17] Google. Espresso. https://google.github.io/android-testing-support-library/docs/espresso/index.html.

[18] Google. Robolectric. http://robolectric.org/.

[19] Google. Android Lint. https://developer.android.com/studio/write/lint.html, 2016.

[20] Google. EarlGrey: iOS UI Automation Test Framework. https://github.com/google/EarlGrey, 2016.

[21] Google. Get started with accessibility scanner, Nov. 2016. https://support.google.com/accessibility/android/answer/6376570.

[22] Google. Google TalkBack. https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback, 2016.

[23] Google. Making apps more accessible, Feb. 2016. https://developer.android.com/guide/topics/ui/accessibility/apps.html.

[24] Google. Testing you app's accessibility, Jan. 2017. https://developer.android.com/training/accessibility/testing.html.

[25] Google. Ui automator, Sept. 2017. https://developer.android.com/training/testing/ui-automator.html.

[26] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

[27] G. Inc. Gradle Build Tool. https://gradle.org, 2017.

[28] KIF. KIF: Keep it functional - an ios functional testing framework. https://github.com/kif-framework/KIF, 2016.

[29] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[30] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin. Amc: Verifying user interface properties for vehicular applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 1–12, New York, NY, USA, 2013. ACM.

[31] F.-D. Limited. F-Croid Catalogue of Free and Open Source Android Apps. https://f-droid.org/, 2017.

[32] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 1013–1024, 2014.

[33] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 224–234. ACM, 2013.

[34] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 599–609. ACM, 2014.

[35] I. Malý, J. Bittner, and P. Slavík. Using annotated task models for accessibility evaluation. In *Int. Conference on Computers Helping People with Special Needs (ICCHP)*, pages 315–322. Springer Berlin Heidelberg, 2012.

[36] J. Mankoff, H. Fait, and T. Tran. Is your web page accessible?: A comparative study of methods for assessing web page accessibility for the blind. In *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 41–50. ACM, 2005.

[37] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for Android applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 94–105, 2016.

[38] H. Muccini, A. Di Francesco, and P. Esposito. Software testing of mobile applications: Challenges and future research directions. In *Int. Workshop on Automation of Software Test (AST)*, pages 29–35. IEEE Press, 2012.

[39] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: An innovative tool for automated testing of GUI-driven software. *Automated Software Eng.*, 21(1):65–105, 2014.

[40] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1):62–66, 1979.

[41] World Bank. Main report - world report on disability.

[42] Y. Yesilada, G. Brajnik, and S. Harper. Barriers common to mobile and disabled web users. *Interact. Comput.*, 23(5):525–542, 2011.

[43] S. Zein, N. Salleh, and J. Grundy. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software (JSS)*, 117:334 – 356, 2016.