Journal of Software Engineering Research and Development, 2022, 10:8, doi: 10.5753/jserd.2022.2133

Accessibility Mutation Testing of Android Applications

Henrique Neves da Silva [**Federal University of Paraná** | ***henriqueneves@ufpr.br***]
Silvia Regina Vergilio [**Federal University of Paraná** | ***silvia@inf.ufpr.br***]
André Takeshi Endo [**Federal University of São Carlos** | ***andreendo@ufscar.br***]

Abstract

Smart devices and their apps are present in many everyday activities and play an important role for people with some disabilities. However, making apps more accessible is still a challenge for developers. Automatically accessibility testing tools can help in this task but present some limitations. They produce reports on accessibility faults, which usually cover only a subset of the app because they are dependent on the test set available. In order to help in the improvement and/or assessment of test suites generated, as well as contribute to increasing the performance of accessibility testing tools, this work introduces a mutation testing approach. The approach includes a set of mutant operators derived from faults corresponding to the negation of the WCAG standard's principles and success criteria. It also includes a process to analyse the mutants regarding the original app. Evaluation results with 7 open-source apps show the approach is applicable in practice and contributes to significantly improving the number of faults revealed by the test suites accompanying the apps.

Keywords: *Mobile Apps, Mutation Testing, Accessibility*

1 Introduction

In the last decade, we have observed a growing number of smartphones and studies show this number is expected to increase even more in the next years (Cisco, 2017). Smart devices and their apps have become a key component in people's daily lives. This is not different for people with some disabilities. For instance, people with some visual impairment have relied on smartphones as a vital means to foster independence in carrying out various tasks, such as understanding text document structure, communicating through social media apps, identifying products on supermarket shelves, and moving between obstacles (Acosta-Vargas et al., 2020).

World Health Organization (WHO) estimated that more than one billion people, which is around 15% of the world's population, are affected by some form of disability (Hartley, 2011). Then, it is fundamental to engineer software so that all the advantages of technology are accessible to every individual. Mobile accessibility refers to making websites and apps more accessible to people with disabilities when using smartphones and other mobile devices (W3C, 2019). Progress has been made with accessibility because of mandates from government regulations (e.g., U.S. Section 508 of Rehabilitation Act), standards (such as the British Broadcast Corporation Standards, Brazilian Accessibility Model, and Web Content Accessibility Guidelines), widespread industrial awareness, technological advances, and accessibility-related lawsuits (Yan and Ramachandran, 2019). However, developers still have the challenge of providing more accessible software on mobile devices. According to Ballantyne et al. (2018), much of the research on software accessibility is dedicated to the Web and its sites (Grechanik et al., 2009; Wille et al., 2016; Abuaddous et al., 2016); even though there is a recurring effort on the accessibility of mobile apps (Vendome et al., 2019). Moreover, studies point to the lack of adequate tools, guides and policies to design, evaluate, and test the accessibility in mobile apps (Acosta-Vargas et al., 2020).

Automated accessibility testing tools are usually based on existing guidelines. One of the most popular standards is the WCAG (W3C's Web Content Accessibility Guideline) (Kirkpatrick et al., 2018) guide. The WCAG guide covers recommendations for people with blindness and low vision, deafness and hearing loss, limited movement, cognitive limitations, speech and learning disabilities. WCAG encompasses several guidelines, each one related to different success criteria, grouped into four accessibility principles. Some tools produce, given a set of executed test cases, a report of accessibility violations for the app. Examples of these tools are Accessibility Google Scanner (Google, 2020), Espresso (Google, 2018), A11y Ally (Toff, 2018), and MATE (Eler et al., 2018). They can perform static or dynamic analysis (Silva et al., 2018).

A limited number of violations can be checked by static tools, but dynamic analysis tends to be more costly. Another limitation is that the accessibility faults checked by tools are limited by the test cases used. They cover only a subset of the app due to weak test scripts or limited input test data generation algorithms (Silva et al., 2018). Tools generally used for test data generation such as Monkey (Moher et al., 2009), Sapienz (Mao et al., 2016), Stoat (Su et al., 2017) and APE (Gu et al., 2019), are focused on functional behavior, code coverage or crashes. In this sense, this work hypothesizes that a mutation approach specific to accessibility testing can help in the improvement and/or assessment of test suites generated and contribute to increasing the performance of accessibility testing tools.

The idea behind mutation testing is to derive versions of the program under test P, called mutants. Each mutant describes a possible fault, and is produced by a mutation operator (Jia and Harman, 2011). The objective is to generate test cases capable of distinguishing P from its mutants, that is, that when executed with each mutant m produces a different output from the output of P. If the P's result is correct, it is free from the fault described by m. If the output is differ-

ent, m is said killed. At the end, a measure called mutation score is calculated, related to the number of mutants killed. This measure can be used to design test cases, or to evaluate the quality of an existing test suite, and consider whether a program has been tested enough.

Mutation testing has been proved to be effective in different domains and contexts (Jia and Harman, 2011). More recently, it has been used in the test of non-functional properties such as performance regarding execution time (Lisper et al., 2017) and energy consumption (Jabbarvand and Malek, 2017). There are some initiatives exploring mutation testing of Android apps (Wei, 2015; Deng et al., 2015; Jabbarvand and Malek, 2017; Luna and El Ariss, 2018; Escobar-Velásquez et al., 2019). But these works are not focused on accessibility testing.

Given the context and motivation described above, this paper introduces a mutation approach for the accessibility test of Android apps. The underlying fault model is related to the non-compliance with WCAG principles and success criteria. We propose a set of 6 operators that remove some selected code elements, the most commonly used in the apps, and whose absence may imply accessibility violations. We also define a mutant analysis process that uses tools' accessibility reports to distinguish killed mutants. The process is implemented using the reports produced by Espresso Google (2018), and evaluated with 7 open-source apps. The results show our approach is applicable in practice and contributes to improving the quality of the test suites accompanying the selected apps. We observe a significant improvement regarding the number of faults revealed by using the mutant-adequate test suites.

In this way, the present work introduces a mutation approach that encompasses a set of mutant operators and a mutation process implemented by a tool. The approach *(i)* can be used as a criterion for test data generation and/or assessment, helping developers measure the quality of their test suites or to generate tests from an accessibility perspective; *(ii)* can be explored to evaluate the accessibility tools available in the market and in academia; and *(iii)* contributes to the emergent area of mutation testing for non-functional properties, and represents the first step to allow accessibility mutation testing, serving as basis to direct future research and encourage the academic community to create tools that further explore this field of research.

The remainder of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 introduces our mutation testing approach. Section 4 details the evaluation and its main results. Section 5 discusses the threats to validity, and Section 6 concludes the paper.

2 Related Work

Related work can be classified into two main categories: mutation testing of apps (Section 2.1) and accessibility testing (Section 2.2).

2.1 Mutation testing of Android Apps

In the literature, there are some mutation approaches for Android apps. Deng et al. (2015) define 4 classes of mutation operators specific to the Android context. The proposed workflow differs from the traditional mutation test process. Once the mutants are generated, it is necessary to install each mutant m on the Android emulator. The test cases are implemented through frameworks Robotium (Reda, 2019) or JUnit (Gamma and Beck, 2019). While Deng's approach requires the app source code, Wei (2015) proposes muDroid, a tool that requires only the APK file of the app.

Linares-Vásquez et al. (2017) define a list of 38 mutation operators, implemented by the tool MDroid+ (Moran et al., 2018). First, a static analysis of Java code using Abstract Syntactic Trees (AST) is performed to find a Potential Fault Profile (PFP) that describes a source code location that can be changed by an operator. PFPs are used to apply the transformation corresponding to each operator in the Java code or XML file. MDroid+ creates a clone of the Android project and applies a single mutation to a PFP specified in the cloned project, resulting in a mutant. Finally, a report is generated associating the name of the created clone with the applied operator. The tool does not offer a way to compile and execute the mutants, nor does it calculate the mutation score.

In a follow-up study, Escobar-Velásquez et al. (2019) introduce MutAPK that requires as input the APK of the Android app and implements the same operators of MDroid+ (Linares-Vásquez et al., 2017; Moran et al., 2018). The corresponding implementation considers SMALI representation. Like MDroid+, MutAPK does not include a mutant analysis strategy. Both allow the creation of customized mutation operators.

Some works have explored aspects of a specific nature within the Android platform. The Edroid tool (Luna and El Ariss, 2018) implements 10 mutation operators oriented to vary configuration files and GUI elements. The analysis of the mutants is done manually. If the mutant's UI components are distinguished from the original, the mutant is classified as dead.

μDroid is a mutation tool to identify energy-related problems (Jabbarvand and Malek, 2017). The tool implements a total of 50 mutation operators corresponding to 28 classes defined as energy consumption anti-patterns. μDroid has a fully automated mutation testing process. While the test is performed in the original app, energy consumption is monitored. When the test is executed on the mutant, the energy consumption of the original app is compared to that of the mutant. If the screening is different enough, the mutant is considered dead.

Most tools may be extended to have integrated support for the mutation testing process, mainly automatic mutant execution and analysis. Most of them generate mutants and do not offer automatic support for the analysis of the mutant output, which is mainly conducted manually. In addition, there are some initiatives exploring mutation testing of apps for non-functional properties, such as energy consumption, but they do not address accessibility faults. Based on elicited results about mutation testing of mobile apps (Silva et al., 2021), and as far we are concerned, there is not a mutation approach for

mobile accessibility testing and evaluation.

2.2 Accessibility evaluation of Android Apps

There are few studies on the accessibility assessment of mobile apps. This small amount of studies is due to the lack of adequate tools, guides, and policies to evaluate apps (Acosta-Vargas et al., 2020; Eler et al., 2018). Such guides are generally used as oracles to check whether the app meets accessibility requirements during accessibility evaluation that can be conducted manually or by automated tools. Below, we present some works that analyse those guides and report the main accessibility problems, as well as automated tools that take them into consideration.

Ballantyne et al. (2018) compile a super-set of guides and normalize them to eliminate redundancy. The result lists 11 categories of testable accessibility elements: Text, Audio, Video, GUI Elements, User Control, Flexibility and Efficiency, Recognition instead of Recalling, Gestures, System Visibility, Error Prevention, and Tangible Interaction. Damaceno et al. (2018) perform a similar mapping that identifies 68 problems associated with different aspects of the interaction of people with visual impairments on mobile devices. These problems are mapped into 7 groups: Buttons, Data Entry, Gesture-based interaction, Screen size, User feedback, and Voice command. The group with more problems is related to the interaction made of formal gestures.

Vendome et al. (2019) elaborate taxonomy of accessibility problems by mining 13,817 Android apps from GitHub. The authors observe that 36.96% of the projects did not have elements with descriptive label attributes, and only 2.08% imported at least one accessibility API. The main categories listed in the fault model are: support for visual limitation, support for motor limitation, hearing limitation, and other aspects of accessibility.

Alshayban et al. (2020) present the results of a large-scale study to understand the accessibility from three complementary perspectives: app, developers, and users. First, they analyze the prevalence of accessibility violations in over 1,000 Android apps. Then they investigate the developer sentiments through a survey. In the end, they investigate user ratings and app popularity. Their analysis revealed that inaccessibility rates for apps developed by big companies are relatively similar to inaccessibility rates for other apps.

The works of Acosta-Vargas et al. (2019, 2020) evaluate the use of WCAG 2.1 and the Accessibility Google Scanner, a tool that suggests accessibility improvements for Android apps. The authors conclude that the WCAG guide achieves digital inclusion on mobile platforms. However, the accessibility problems must be fixed before the application goes into production and they recommend the use of WCAG throughout the development cycle.

The most recent version of WCAG 2.1 includes suggestions for web access via a mobile device (Kirkpatrick et al., 2018). WCAG principles are grouped into 4 categories: (i) Perceivable, that is, “the information must be presentable to users in ways they can perceive”; (ii) Operable, “User interface components and navigation must be operable.”; (iii) Understandable, “Information and the operation of user interface must be understandable.”; and (iv) Robust, “Content must be robust enough that it can be interpreted by a wide variety of user agents, including assistive technologies”. These principles are the core tenets of accessibility. To follow the accessibility principles, we must achieve the success criteria defined within their respective guideline and principle.

Automated tools commonly use the WCAG success criteria as testable statements to check for guideline violations. They can perform static or dynamic analysis (Silva et al., 2018). Static analysis can quickly analyze all assets of an app (Google, 2018), but they cannot find violations that can only be detected during runtime (e.g., low color contrast). In contrast, dynamic analysis tends to be time consuming. In this sense, Eler et al. (2018) define a set of accessibility criteria and implemented MATE (Mobile Accessibility TEsting), a tool that automatically explores and verifies the accessibility of mobile apps.

Developers can also manually assess accessibility properties using the Google Scanner (Google, 2020). It allows testing apps and gets suggestions on how to improve accessibility (to help those who have limited vision, speech, or movement). First, the app is activated, then it displays the main handling instructions. Finally, with the mobile app running, Google Scanner highlights the GUI element on the screen and what accessibility property it has not fulfilled.

The A11y Ally app (Toff, 2018) checks the accessibility of the running app. From its integration via the command line, a11y generates a JSON file at the end of its execution. This file contains the list of GUI elements and which accessibility criteria have been violated. The framework Espresso (Google, 2018) allows the recording of automated tests that assess the accessibility of the mobile app. The accessibility of the GUI element, or only widget, will be checked if the test action triggers/interacts with the widget in question.

The tools for accessibility testing and evaluation present some limitations. The most noticeable one is that the kind and number of accessibility violations determined by the tools are dependent on the test set used to execute the app and produce the reports. In this sense, the use of mutants describing potential accessibility faults can guide the test data generation and help in the improvement or assessment of an existing test set regarding this non-functional property.

3 A Mutation Approach for Accessibility Testing

This section introduces our approach, and describes its main elements, which are usually required for any mutation approach: *(i)* the underlying fault model, related to accessibility faults; *(ii)* the mutation operators; *(iii)* the mutation testing process, adopted to analyze the mutants; and *(iv)* automation aspects, essential to allow the use of the approach in practice.

3.1 Fault Model

In this stage, we searched the literature for different accessibility guides that establish good practices and experiments that used them (see Section 2.1). In general, a guide summarizes the main recommendations for making the presented content of the mobile app more accessible. As a result of

our search, we observe that the WCAG guide was adopted as a reference to build mobile accessibility guides such as eMAG (Brazilian Government, 2007), List of Accessibility Guidelines for Mobile Applications (Ballantyne et al., 2018), BBC Mobile Accessibility Guideline (BBC, 2017), and SiDi Accessibility Guideline (SejaSidier, 2015). In this way, the WCAG guide was chosen due to the following reasons: i) as mentioned before, it encompasses success criteria written as testable statements; ii) it is constantly updated and a new version of the guide maintains compliance with its previous one; and iii) it has been considered by many authors as the most popular guide (Acosta-Vargas et al., 2019, 2020). Once the success criteria are known, we can start building a fault model by negating these criteria. An unsatisfied criterion may imply one or more accessibility faults, as exemplified in Table 1.

Table 1. Negating WCAG success criteria

Principle	Success criterion	Success criterion denial
Perceivable	Content description for non-text elements	Absence of content descriptions
Operable	Recommended touch area size	Not recommended touch area size
Understandable	Labels or instructions	Absence of labels or instructions
Robust	Status messages	Absence of status messages

As observed in Table 1, the denial of the criterion "Labels or instructions" causes one or more faults related to the absence of a label. Within Android's mobile development, different code elements characterize the use of a label for a GUI element. These code elements can be either XML attributes or Java methods. For instance, one way to satisfy the success criterion "Labels or instructions" is setting the XML attributes `:hint` and `:labelFor`, or using the Java methods `setHint` and `setLabelFor`. Such elements are the key to the generation of mutants, in order to capture the faults of our model. In this way, more than one mutation operator can be derived from the negation of a criterion, such as "Labels or instructions". Each mutation operator, in its turn, can be applied to more than one element in the code, generating distinct mutants.

To select the code elements and propose the mutation operators of our approach, we refer to the work of Silva et al. (2020). This work maps the WCAG principles and success criteria to code elements of native Android API, and analyzes the prevalence of the mapped elements in 111 open source mobile apps. The study identifies code elements that impact accessibility, and shows that apps which adopt different types of code elements tend to have a smaller density of accessibility faults. This means that code elements associated with WCAG are related to accessibility faults and justify mutation operators based on these code elements.

3.2 Mutation Operators

The main objective in defining the accessibility mutation operators is to make sure that the test suite created by the tester exploits all, or at least most, of the app GUI elements, as well as check the correct use of the code elements related to the accessibility success criterion. In this way, the operators can be used to guide the generation of test cases or to assess the quality of existing ones. To this end, and following the work of Silva et al. (2020), we selected a set E of code elements, the most adopted in the apps, to propose an initial set of operators. These operators are defined considering aspects of Android apps' accessibility and can be improved in the future, by adding other code elements and success criteria. The selected code elements are presented in Table 2; they correspond to the most used ones in the apps for each principle (Silva et al., 2020). The table also shows the corresponding mutation operator.

The `labelFor` element is a label that accompanies the View object. It can be defined via the XML file or the Java language. In general, it provides a description and exploration labels for some screen elements. The `hint` element is a temporary label assigned to editable fields only. It is necessary for TalkBack, or any other screen reader, to correctly report what information the app needs. We can set or change TextView font size with the element `textSize`. Recommended dimension type for text is "sp" for scaled-pixels (e.g., 15sp). The element `inputType` specifies the input type for each text field in order for the system to display the appropriate soft input method (e.g., an on-screen keyboard). The app, by default, looks for the closest element to receive the next focus. The next element is not always the most logical. In these cases, we need to give the app custom navigation. We can define the next view to focus on using the code element `nextFocusDownId`. The element `importantForAccessibility` describes whether or not this view is important for accessibility. If the value is set with "yes", the view fires accessibility events and is reported to accessibility services (e.g., TalkBack) that query the screen.

The idea of the operators is to remove the corresponding code element $e \in E$ when present. We opted for statement deletion operators, as previous studies gave evidence that such operators produce fewer yet effective mutants (Delamaro et al., 2014). For each code element removed, we have a unique generated mutant. Table 3 presents examples of applying the operators. Snippets of code are presented and the ones to be removed are preceded by "–". It is important to emphasize that if a mutation operator can not be applied to the app source code, this may indicate that the project/developer team has low priority on accessibility. Now, imagine that the developer has taken care to define the accessibility code elements in the app. Even if they are defined, it is very important to ensure that the test set includes a test that performs an action and interacts with the corresponding GUI element and check they are defined properly.

3.3 Mutation Process

The testing process for the application of the proposed operators is depicted in Figure 1. It encompasses three steps. The first one is the mutant generation using the accessibility mutation operators defined. This step produces a set of mutant apps M. In the second step, the original app and the mutants in M are executed with test set T, which can be designed

Table 2. Selected code elements and corresponding WCAG principles and success criteria.

Principle	Success criteria	Code Elements		Mutation Operator
		XML Attributes	Java Methods	
Perceivable	Resize Text	`:textSize`	`setTextSize`	Missing textSize
	Identify Input Purpose	`:inputType`	`setInputType`	Missing inputType
Operable	Keyboard; Focus Order	`:nextFocusDownId`	`setNextFocusDowndId`	Missing nextFocusDownId
Unders-tandable	Label or Instructions	`:labelFor`	`setLabelFor`	Missing labelFor
	Label or Instructions	`:hint`	`setHint`	Missing hint
Robust	Status Messages	`:importantForAccessibility`	`setImportantForAccessibility`	Missing importantForAccessibility

Table 3. Mutation Operator Description

Mutation operator	Code context example	
	XML Attribute	Java Method
MTS - Missing textSize	<TextView –android:textSize="10sp" 	– textView.setTextSize(size);
MIT - Missing inputType	<EdiText –android:inputType="numberPassword" 	– ed1.setInputType(.TYPE_CLASS);
MNFD - Missing nextFocusDownId	<TextView –android:nextFocusDown="@id.ed2" 	– edit.setNextFocusDownId(R.id.ed2);
MLF - Missing labelFor	<TextView –android:labelFor="EditText1" 	– textView.setLabelFor(editText);
MH - Missing hint	<EditText –android:hint="foo" 	– myEditText.setHint("foo");
MIA - Missing importantForAccessibility	<EditText –android:importantForAccessibility="yes" 	– view.setImportantForAccessibility(...);

with the tester's preferred strategy. However, for the mutant analysis our process requires that T is implemented and executed by using an accessibility checker tool, such as the ones reported in Section 2.2. The third step, mutant analysis, allows calculating the mutation score by comparing the accessibility reports produced by an accessibility checker for the original and mutant apps. If the accessibility logs differ, that is, different accessibility faults are encountered the mutant can be considered dead. The accessibility report generated by Espresso contains some temporal information that may cause a non-deterministic output. To correct this, we post-process the output so that only the essential information is taken into account, namely the code element ID and its reported accessibility issue.

Therefore, if the original app's accessibility log is the same as that of the mutant app, resulting in a live mutant, the test suite probably needs to be revised and improved. If the score is not satisfactory, the tester can add new test cases or modify existing ones in T so that more mutants are killed.

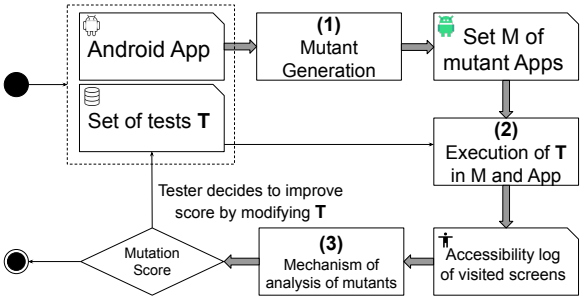

Figure 1. Testing process of the proposed approach

3.4 Implementation

To evaluate and use our approach, we implemented a prototype tool named AccessibilityMDroid. It receives as input the source code of the Android app under test. AccessibilityMDroid implements the proposed operators by extending MDroid+ (Moran et al., 2018), which are used for mutant generation (Step 1). To build and execute the test, as well as to produce the accessibility log (Step 2), the Espresso framework is used. We chose tests implemented with Espresso because it is the default framework for GUI testing in Android Studio and includes embedded accessibility checking. As T is executed, the AccessibilityCheck class allows us to check for accessibility faults. In the end of the run, Espresso generates a log of the accessibility problems used in Step 3. The tool compares the log automatically, and a list of mutants killed is produced.

To illustrate our approach we use a sample app built with Android Studio. A piece of code for this app is presented in Figure 2. With the application of operator MH (Missing hint), which removes from the GUI element the hint code element, Line 22 (in red) disappears in the mutant m.

```
14 <EditText
15     android:id="@+id/nickname"
16     android:layout_width="0dp"
17     android:layout_height="wrap_content"
18     android:layout_marginStart="24dp"
19     android:layout_marginTop="96dp"
20     android:layout_marginEnd="24dp"
21     android:inputType="text"
22     - android:hint="Nickname"
23     android:selectAllOnFocus="true"
24     app:layout_constraintEnd_toEndOf="parent"
25     app:layout_constraintStart_toStartOf="parent"
26     app:layout_constraintTop_toTopOf="parent" />
```

Figure 2. A mutant generated by operator MH

```
@Test
public void loginTest() {
    var appCompatEditText = onView(allOf(
      withId(R.id.username),
      childAtPosition(allOf(withId(R.id.container),
      childAtPosition(withId(android.R.id.content), 0)),
           1),
      isDisplayed()));

    appCompatEditText.perform(replaceText("email"),
         closeSoftKeyboard());

    var appCompatEditText2 = onView(allOf(
      withId(R.id.password),
      childAtPosition(allOf(withId(R.id.container),
      childAtPosition(withId(android.R.id.content), 0)),
           2),
      isDisplayed()));

    appCompatEditText2.perform(replaceText("123456"),
         closeSoftKeyboard());

    var appCompatEditText3 = onView(allOf(
      withId(R.id.password), withText("123456"),
      childAtPosition(allOf(withId(R.id.container),
      childAtPosition(withId(android.R.id.content), 0)),
           2),
      isDisplayed()));

    appCompatEditText3.perform(pressImeActionButton());
}
```

Figure 3. Test case using Espresso

```
AppCompatEditText{id=2131230902,res-name=nickname}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).

AppCompatEditText{id=2131230902,res-name=nickname}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).

AppCompatEditText{id=2131230917,res-name=password}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).

AppCompatEditText{id=2131230917,res-name=password}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).

AppCompatEditText{id=2131230917,res-name=password}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).
```

Figure 4. Accessibility log for the original app

```
@Test
public void loginTest() {
+ onView(withId(R.id.nickname)).perform(typeText("nick"),
+ closeSoftKeyboard());
   var appCompatEditText = ...
}
```

Figure 5. Changed test

Suppose that for this app, a test, as depicted in Figure 3, is available. When T is executed with Espresso on mutant m (Step 2), a log is generated. This log is compared to the log generated by executing T in the original app (Step 3). From the difference between the two accessibility logs, it is possible to determine the mutant's death. In this case, T was not enough to show the difference between the original app and the mutant. As both produce the same log in Figure 4, the mutant is still alive. The tester now tries to improve T and realizes that existing tests do not interact with one of the

```
+ AppCompatEditTextid=2131230902,res-name=nickname: View
      is missing speakable text needed for a screen
      reader
AppCompatEditText{id=2131230902,res-name=nickname}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).

AppCompatEditText{id=2131230902,res-name=nickname}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).

AppCompatEditText{id=2131230917,res-name=password}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).

AppCompatEditText{id=2131230917,res-name=password}: View
      falls below the minimum recommended size for touch
      targets. Minimum touch target size is 48x48dp.
      Actual size is 331.4x45.0dp (screen density is 2.6).
```

Figure 6. Accessibility log for the mutant m

app's input fields. After changes in T (illustrated in Figure 5), Step 2 is executed again and the log for m is now the one in Figure 6; it differs from the original one in the first line. By employing a similar procedure to kill accessibility mutants, T achieves a higher mutation score, covers more GUI elements, and potentially reveals other accessibility faults.

4 Evaluation

The main goal of the proposed operators is to serve as a guide for the evaluation and improvement of test suites regarding accessibility faults. To evaluate these aspects properly, as well as our implementation using Espresso, we formulated three research questions as follows.

RQ1: How is the applicability of the accessibility mutation operators? This question aims to investigate if the proposed operators and processes are applicable in practice. To answer this question, we evaluate the approach's application cost by analysing the number of mutants generated by each operator, as well as the number of required test cases.

RQ2: How adequate are existing test suites with respect to the accessibility mutation testing? This question evaluates the use of the proposed operators as an evaluation criterion. They are used for quality assessment of the test suites accompanying the selected open source apps with respect to accessibility. To this end, we analyse the ability of existing tests to kill the mutants generated by our approach.

RQ3: How much do the mutation operators contribute to revealing new accessibility faults? This question looks at the effectiveness of mutant-adequate test suites when revealing accessibility violations.

4.1 Study Setup

We sampled open source apps from F-droid[1], last updated in 2019/2020, containing Espresso test suites. We refer to the test suite accompanying the project as T. We removed apps that failed to build and whose tests were not compatible with the accessibility checking feature. The replication package is available at: `https://osf.io/vfs2d/`.

The seven apps are: `AlarmClock`, an alarm clock for Android smartphones and tablets that brings a pure alarm experience; `AnyMemo`, a spaced repetition flashcard learning software; `Authorizer`, a password manager for Android; `Equate`, a unit converting calculator; `KolabNotes`, a note taking app; `Piwigo`, a photo gallery app for the web; and `PleesTracker`, a sleep tracker.

For each app, we used AccessibilityMDroid to generate mutants, run T, produce the accessibility logs to each mutant, and compare those with the original log. In this way, we obtained the set of mutants killed by T. After this, we manually inspected the alive mutants and realized that many times, some of the test cases in T exercised the mutated code, but they produced no difference in the log due to some Espresso limitations (e.g., a limited set of accessibility criteria that will be detected and printed in the accessibility log). In this case, we marked the corresponding mutant as covered. Other mutants were marked as "unreachable" since their mutations are related to widgets that are not reachable in the app (e.g., dead code). So, we counted the number of generated, killed, covered, and unreachable mutants by T.

Then, we extended T so that all mutants were killed or at least covered. We refer to this extended test suite as xT. The inclusion of a test case was conducted in the following way: *(i)* pick an alive mutant (not covered, not killed by T); *(ii)* manually record a test that exercises the mutation using Espresso Test Recorder in Android Studio, and if needed, refactor the test code to make it repeatable[2]; *(iii)* analyze if the mutant is killed by the new test, if not mark it as covered. The mutants information was collected again for xT.

As cost indicators, we collected the number of tests of a test suite $TC(T)$, and its size, given by the number of lines of test code $LoC(T)$. As for effectiveness, we counted per test suite the number of accessibility faults reported by the Espresso accessibility check.

Table 4 shows the information on the seven selected apps. `Authorizer` is the app with the greatest value of LoC (28,286), while `AnyMemo` has 30 activities (#Act.). `AlarmClock` is the app with the smallest number of LoC: 1,349, and `Equate` has only 2 activities. The table also shows the number of test cases (#TC) and LoC for the original set T and the extended one xT. Notice that `AlarmClock` has 41 tests and 1,068 lines of test code ($LoC(T)$). `Kolabnotes` has only one test, yet `AnyMemo` has the smallest $LoC(T)$ (76). Concerning xT, `AlarmClock` and `Authorizer` require more tests (both 43) and more $LoC(xT)$ (1,341 and 1,700, respectively). `PleesTracker` has the smallest number of test cases (5) and $LoC(xT)$ (345). However, `Authorizer` required more additional test cases, 32, while `Piwigo` only one.

[1] `https://www.f-droid.org`

[2] The code generated by Espresso Test Recorder may be too specific and fails in re-runs.

Table 4. Selected apps

App*	LoC	#Act.	#TC(T)	LoC(T)	#TC(xT)	LoC(xT)
`AlarmClock`	1,349	5	41	1,068	43	1,341
`AnyMemo`	19,751	30	3	76	13	932
`Authorizer`	28,286	7	11	652	43	1,700
`Equate`	5,826	2	6	511	9	709
`Kolabnotes`	11,025	9	1	494	6	884
`Piwigo`	4,744	7	8	408	9	579
`PleesTracker`	1,868	5	2	89	5	345

* The app's name is a clickable link to the GitHub project.

4.2 Analysis of Results

Table 5 summarizes the main results of the evaluation and is used in this section to answer our RQs. This table shows the number of mutants that were generated (columns G), killed by some test (columns K), covered but alive (columns C), and unreachable (columns U).

Notice that the results are shown for 4 out of 6 operators described in Table 3; operators MLF and MNFD did not generate any mutant for the selected apps. For each app, two rows are presented, one for the results obtained by T and the other for xT. The last four columns list the total for all operators, while the last rows bring the total for all apps.

For instance, for the app `AnyMemo` the operator MTS generated 64 mutants, 11 unreachable. The test set T was not capable of killing any mutant but covered 14. The set xT covered 52; that is, 38 additional mutants could be covered. Considering all operators, only one mutant could be killed by xT, and 70 mutants were covered out of 84 generated mutants. For this app, four mutants change a screen that is reached only when integrated with a third-party app. As exercising these mutants would require other tools beyond Espresso, we were not able to cover them. However, they can not be classified as unreachable. Because of this, the sum of killed, covered but alive, and unreachable mutants are not equal to the number of generated mutants for this app, as it happens for all of the other ones.

RQ1 – Approach applicability. To answer RQ1, we evaluate the number of mutants generated by each operator. We observe in Table 5 that operator MTS generated more mutants (145 in total), followed by MIT (68), MH (34), and MIA (9). MTS generated mutants for all apps, MIT for 6, and MH for 5 apps. Operator MIA generated mutants only for `Authorizer`. In total, 256 mutants were generated, with `AnyMemo` with more mutants (86) and `Piwigo` with 5. This means that the apps selected contain more code elements associated with the principle Perceivable (operators MTS and MIT), which may indicate: (i) developers are worried about content descriptions for non-text elements more than the principle Robust (operator MIA that generated mutants for only one app) or Operable (operator MNFD that did not generate any mutant); (ii) User Experience (UX) and User Interface (UI) documents include a more significant amount of code elements of the Perceivable principle in their guidelines.

Operators MIT and MIA generated mutants that were not killed; only one mutant of MTS was killed, and 17 out of 34 mutants generated by MH were killed. The process using Espresso was capable of distinguishing mutants in the great majority generated by removing the code element `:hint`. Analysing alive mutants, we identified 222 as covered, and 12 as unreachable. Unreachable mutants were gen-

Table 5. Summary of the results per operator

Android App		Mutation Operator																Total			
		MTS				MIT				MH				MIA							
		G	K	C	U	G	K	C	U	G	K	C	U	G	K	C	U	G	K	C	U
AlarmClock	T	12	-	9	-	1	-	-	-	1	-	-	-	-	-	-	-	14	-	9	-
	xT		-	12			-	1			1	-			-	-			1	13	
AnyMemo	T	64	-	14	11	22	-	-	-	-	-	-	-	-	-	-	-	86	-	14	11
	xT		1	52			-	18			-	-			-	-			1	70	
Authorizer	T	18	-	1	-	27	-	3	-	18	-	3	-	9	-	2	-	72	-	9	-
	xT		-	18			-	27			6	12			-	9			6	66	
Equate	T	3	-	-	-	2	-	-	-	2	1	-	1	-	-	-	-	7	1	-	1
	xT		-	3			-	2			1	-			-	-			1	5	
Kolabnotes	T	23	-	8	-	13	-	3	-	12	-	-	-	-	-	-	-	48	-	11	-
	xT		-	23			-	13			8	4			-	-			8	40	
Piwigo	T	1	-	-	-	3	-	3	-	1	1	-	-	-	-	-	-	5	1	3	-
	xT		-	1			-	3			1	-			-	-			1	4	
PleesTracker	T	24	-	8	-	-	-	-	-	-	-	-	-	-	-	-	-	24	-	8	-
	xT		-	24			-	-			-	-			-	-			-	24	
Total	T	145	-	40	11	68	-	9	-	34	2	3	1	9	-	2	-	256	2	54	12
	xT		1	133			-	64			17	16			-	9			18	222	

Number of mutants **G**enerated, **K**illed, **C**overed but alive, **U**nreachable by the original test suite T and the extended one xT.
The Mutation Operators are: **M**issing **t**ext**Size**; **M**issing **i**nput**T**ype; **M**issing **H**int; and **M**issing **i**mportantFor**A**ccessibility.

Table 6. Efforts to build xT

App	#Mutants / app KLoC					A-TC	A-LoC
	MTS	MIT	MH	MIA	Total		
AlarmClock	8.9	0.7	0.7	0.0	10.37	2	273
AnyMemo	3.2	1.1	0.0	0.0	4.35	10	856
Authorizer	0.6	0.9	0.6	0.3	2.58	32	1048
Equate	0.5	0.3	0.3	0.0	1.20	3	198
Kolabnotes	2.0	1.1	1.0	0.0	4.35	5	390
Piwigo	0.2	0.6	0.2	0.0	1.05	1	171
PleesTracker	12.8	0.0	0.0	0.0	12.8	3	256
Average	4.0	0.67	0.4	0.043	5.42	8	456

A-TC stands for the number of test cases added to T to obtain xT.
A-LoC stands for the number of LoC added to T to obtain xT.

erated mainly for AnyMemo and are related to implementation smells like dead code.

For a deeper analysis, Table 6 contains the number of mutants generated by the operator divided by the KLoC of each app. The last two columns present information regarding the effort required to add new test cases so that an accessibility mutant adequate test suite is obtained. The last rows contain min, max and average values. We can see that the operators generate a mean value of 5.42 mutants per KLoC, and, in the worst case, 12.8 for PleesTracker.

Notice that a greater number of mutants is generated for the largest apps in terms of LoC and number of activities: AnyMemo, Authorizer and Kolabnotes. Given the fact that the proposed operators only remove code elements, the number of mutants tends to be equal to the number of existing elements associated to the accessibility WCAG success criteria.

Due to this characteristic, it is unlikely that the operators generate equivalent mutants. This is an advantage, because the identification of such mutants is usually costly. Moreover, we have not found either stillborn or trivial mutants. The first ones are mutants that do not compile, and the second ones are mutants that crash in the initialization. We also measured the effort of adding new test cases, considering the values in Table 4. As Table 6 shows, Authorizer demanded more effort required 32 additional tests (with 1,048 A-LoC), followed by AnyMemo: which required 10 additional tests (with 856 A-LoC); and Kolabnotes: 5 tests (390 A-LoC). These apps are the greatest in terms of size.

> *Response to RQ1:* The number of mutants is related to the size of the app, mainly to the number of GUI elements, and code elements associated with the accessibility success criteria. Operators MTS and MIT, related to the principle Perceivable, produce more mutants, while no mutant is generated for operator MNFD, related to the Operable principle. Moreover, we did not observe any stillborn, trivial, or equivalent mutants.

Implications: The operators are deletion style and depend on the use of accessibility-related code elements. The number of generated mutants grows proportionally to the number of accessibility code elements used in the app. Operators MTS and MIT generated more mutants, which may indicate that code elements related to the principle Perceivable are the most used in the app selected. Our set of operators represents a first proposal, and we intend to improve the set with other kinds of operators, that for instance adding or modifying code elements, as well, and other code elements and success criteria could be considered.

The proposed operators do not generate equivalent mutants due to their conception characteristics. We did not observe any stillborn or trivial mutant. This is important, because they imply in cost. These kinds of mutants are very common in the Android mutation testing (Linares-Vásquez et al., 2017).

We observe Espresso's limited ability to detect accessibility faults, and as a consequence, a reduced number of mutants were killed. Because of this other accessibility testing tools should be used in future versions of AccessibilityMDroid. We also intend to implement mechanisms to automatically determine covered mutants. The analysis of dead mutants is a drawback of most mutation testing approaches for Android apps. The great majority do not offer an automatic way to per-

form this task, they do not even provide a way to consider a mutant killed.

RQ2 – Adequacy of existing test suites. RQ2 evaluates the adequacy of the test suites concerning the proposed operators. The answer can shed some light on the quality of the test cases regarding accessibility faults and if the developers are worried about the test of such a non-functional property. To answer this question, Table 7 brings the percentage of mutants killed and covered by T, per app. Unreachable mutants were not considered. On average, the original sets were capable of killing only 5.23% of the mutants. The killed percentage reaches 20% for `Piwigo`, the app with the fewest number of mutants. But this percentage is equal to zero for five apps. The percentage of covered mutants are better, 30.24% on average. The best percentages were achieved by `AlarmClock` (64.3%) and `Piwigo` (60%). The other five apps achieved a percentage lower than 35%.

Table 7. Adequacy results of original test suites

App	Killed	Covered
`AlarmClock`	0.0%	64.3%
`AnyMemo`	0.0%	18.67%
`Authorizer`	0.0%	12.5%
`Equate`	16.67%	0%
`Kolabnotes`	0.0%	22.91%
`Piwigo`	20%	60%
`PleesTracker`	0.0%	33.33%
Average	5.23%	30.24%

Response to RQ2: The existing test suites of the studied apps killed or covered only a small fraction of the accessibility-related mutants. In other words, they had a low mutation score.

Implications: In general, there are opportunities to improve the quality of GUI tests in mobile apps. While code coverage and mutation testing have better support at the unit test level, more tool support is required at GUI level. As the accessibility mutants demand better test coverage at GUI level, the results herein presented helped to expose those weaknesses.

RQ3 – Accessibility faults. By answering RQ2, we observe that the existing tests obtained a small coverage of accessibility mutants, and new tests are required to obtain adequate test suites. However, it is important to know if such additional tests and efforts improve the test quality in terms of accessibility faults revealed. RQ3 aims to answer this question.

Table 8 shows the number of accessibility faults pointed by Espresso when the original (T) and extended (xT) test sets are used; the last column also shows the percentage of improvement. For T, `AlarmClock` has more accessibility faults (126), while `PleesTracker` has only 2 faults. On average we have 45.28 accessibility faults per app. Concerning the mutant-adequate test suite xT, `Piwigo` has more faults (447); `PleesTracker` presented the best percentage of improvement (3,650%). But, the smallest percentage of improvement was obtained for `AlarmClock`. On average xT revealed 186.4 accessibility faults. The improvements varied from 3.2 to 3,650%.

Table 8. Accessibility faults detected by T and xT

App	#faults(T)	#faults(xT)	Improv.
`AlarmClock`	126	130	3.2%
`AnyMemo`	24	355	1,479%
`Authorizer`	65	201	209.2%
`Equate`	19	27	42.1%
`Kolabnotes`	43	70	62.8%
`Piwigo`	38	447	1,076.3%
`PleesTracker`	2	75	3,650%
Average	45.28	186.4	931.8%

Response to RQ3: Mutant-adequate test suites contribute to meaningful improvements in the number of accessibility faults detected. On average, the extended test suites improved around 932% the number of accessibility faults revealed in the original test suites.

Implications: The results gave evidence that the use of the mutation operators contributed to an increase in the number of revealed accessibility faults. We anticipate that the quality of the test suite is improved too, besides the accessibility point of view.

5 Threats to Validity

There are some threats to the validity of our study.

Sample selection. It is not easy to guarantee the representativeness of the apps. In addition, the adopted sample has only Android native apps with Espresso test suites. To mitigate this, we selected the apps from F-Droid a diverse set of open-source apps with recent updates. F-Droid has been used in other studies (Mao et al., 2016; Zeng et al., 2016; Gu et al., 2019).

Limited oracle. The mutant analysis strategy is linked to the Espresso tool. However, the proposed approach is also compatible with other tools that monitor the running app and produce accessibility logs like MATE (Eler et al., 2018) and A11y (Toff, 2018); we plan to integrate them in the future.

Manual determination of covered elements. This task was performed manually and is subject to errors. To minimize this threat, this analysis was carefully conducted and double-checked.

Flaws in the implementation. There may be implementation errors in any of tools or routines used in our study, like the MDroid+ extension, Android emulator management, and Espresso.

The number of mutation operators. The set of accessibility mutation operators proposed represents only a fraction of all accessibility violations that can occur in a mobile app. We created this initial deletion set to validate the proposed tool. This set of deletion mutation operators is tested and validated as effective in practice.

6 Concluding Remarks

This paper presented an approach for accessibility mutation testing of Android apps. First, we defined a set of six accessibility mutation operators for Android apps. Then, for an Android app, we generated the mutants. Based on the original test suite, we checked which mutants are killed or at least covered. Following our approach, we extended the original test suite to cover more mutants. The empirical results show that the original test suites cover only a small part of the accessibility-related mutants. Besides, mutant-adequate test suites contribute to meaningful improvements in the number of accessibility faults detected.

As future work, we plan to extend the tool support to handle APK files and commercial apps (closed source). The mutation operators may also be described more generically so that the approach can be extended to include other mobile development languages and frameworks (e.g., Swift, React-Native, Kotlin).

Another direction is to experiment with different oracles (e.g., MATE (Eler et al., 2018)), besides the accessibility check of Espresso we used in this study. Finally, different accessibility mutation operators can be defined, now focused on including and changing code elements.

Acknowledgment

This work is partially supported by CNPq (Andre T. Endo grant nr. 420363/2018-1 and Silvia Regina Vergilio grant nr. 305968/2018-1).

References

Abuaddous, H. Y., Jali, M. Z., and Basir, N. (2016). Web accessibility challenges. *International Journal of Advanced Computer Science and Applications (IJACSA)*.

Acosta-Vargas, P., Salvador-Ullauri, L., Jadán-Guerrero, J., Guevara, C., Sanchez-Gordon, S., Calle-Jimenez, T., Lara-Alvarez, P., Medina, A., and Nunes, I. L. (2020). Accessibility assessment in mobile applications for android. In Nunes, I. L., editor, *Advances in Human Factors and Systems Interaction*, pages 279–288, Cham. Springer International Publishing.

Acosta-Vargas, P., Salvador-Ullauri, L., Perez Medina, J. L., Zalakeviciute, R., and Perdomo, W. (2019). Heuristic method of evaluating accessibility of mobile in selected applications for air quality monitoring. In *International Conference on Applied Human Factors and Ergonomics*, pages 485–495. Springer.

Alshayban, A., Ahmed, I., and Malek, S. (2020). Accessibility issues in android apps: State of affairs, sentiments, and ways forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1323–1334, New York, NY, USA. Association for Computing Machinery.

Ballantyne, M., Jha, A., Jacobsen, A., Hawker, J. S., and El-Glaly, Y. N. (2018). Study of Accessibility Guidelines of Mobile Applications. In *Proceedings of the 17th International Conference on Mobile and Ubiquitous Multimedia*, pages 305–315. ACM.

BBC (2017). The BBC Standards and Guidelines for Mobile Accessibility. `https://www.bbc.co.uk/accessibility/forproducts/guides/mobile`.

Brazilian Government (2007). Accessibility Model in Electronic Government. `https://www.gov.br/governodigital/pt-br/acessibilidade-digital/modelo-de-acessibilidade`.

Cisco (2017). Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017–2022 White Paper - Cisco. `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html`.

Damaceno, R. J. P., Braga, J. C., and Mena-Chalco, J. P. (2018). Mobile device accessibility for the visually impaired: problems mapping and recommendations. *Universal Access in the Information Society*, 17(2):421–435.

Delamaro, M. E., Offutt, J., and Ammann, P. (2014). Designing deletion mutation operators. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 11–20.

Deng, L., Mirzaei, N., Ammann, P., and Offutt, J. (2015). Towards mutation analysis of Android apps. In *Proceedings of the Eighth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW, pages 1–10. IEEE.

Eler, M. M., Rojas, J. M., Ge, Y., and Fraser, G. (2018). Automated Accessibility Testing of Mobile Apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126.

Escobar-Velásquez, Camilo, O.-R., Michael, and Linares-Vásquez, M. (2019). MutAPK: Source-Codeless Mutant Generation for Android Apps. In *2019 IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Gamma, E. and Beck, K. (2019). The new major version of the programmer-friendly testing framework for Java. `https://junit.org`.

Google (2018). Espresso. `https://developer.android.com/training/testing/espresso`.

Google (2018). Improve your code with lint checks. `https://developer.android.com/studio/write/lint`.

Google (2020). Accessibility Scanner. `https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_U`.

Grechanik, M., Xie, Q., and Fu, C. (2009). Creating gui testing tools using accessibility technologies. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 243–250.

Gu, T., Sun, C., Ma, X., Cao, C., Xu, C., Yao, Y., Zhang, Q., Lu, J., and Su, Z. (2019). Practical GUI Testing of Android Applications via Model Abstraction and Refinement. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 269–280. IEEE Press.

Hartley, S. D. (2011). World Report on Disability (WHO). Technical report, WHO and World Bank.

Jabbarvand, R. and Malek, S. (2017). μDroid: an energy-aware mutation testing framework for Android. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 208–219. ACM.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678.

Kirkpatrick, A., Connor, J. O., Campbell, A., and Cooper, M. (2018). Web Content Accessibility Guidelines (WCAG) 2.1. `https://www.w3.org/TR/WCAG21/`.

Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., Bernal-Cárdenas, C., and Poshyvanyk, D. (2017). Enabling Mutation Testing for Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 233–244, New York, NY, USA. ACM.

Lisper, B., Lindstrom, B., Potena, P., Saadatmand, M., and Bohlin, M. (2017). Targeted mutation: Efficient mutation analysis for testing non-functional properties. In *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, (ICSTW)*, pages 65–68.

Luna, E. and El Ariss, O. (2018). Edroid: A Mutation Tool for Android Apps. In *Proceedings of the 6th International Conference in Software Engineering Research and Innovation*, CONISOFT, pages 99–108. IEEE.

Mao, K., Harman, M., and Jia, Y. (2016). Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA. Association for Computing Machinery.

Moher, D., Liberati, A., Tetzlaff, J., and Altman, D. G. (2009). Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement. *BMJ*, 339.

Moran, K., Tufano, M., Bernal-Cárdenas, C., Linares-Vásquez, M., Bavota, G., Vendome, C., Di Penta, M., and Poshyvanyk, D. (2018). Mdroid+: A mutation testing framework for android. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 33–36. ACM.

Reda, R. (2019). RobotiumTech: Android UI Testing. `https://github.com/RobotiumTech/robotium`.

SejaSidier (2015). Guide to the Development of Accessible Mobile Applications. `http://www.sidi.org.br/guiadeacessibilidade/index.html`.

Silva, C., Eler, M. M., and Fraser, G. (2018). 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*, DSAI 2018, page 286–293. ACM.

Silva, H. N., Endo, A. T., Eler, M. M., Vergilio, S. R., and Durelli, V. H. R. (2020). On the Relation between Code Elements and Accessibility Issues in Android Apps. In *Proceedings of the V Brazilian Symposium on Systematic and Automated Software Testing*, SAST.

Silva, H. N., Prado Lima, J. A., Endo, A. T., and Vergilio, S. R. (2021). A mapping study on mutation testing for mobile applications. *Software Testing, Verification Reliability*.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., and Su, Z. (2017). Guided, stochastic model-based GUI testing of android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, Paderborn, Germany, September 4-8*, pages 245–256.

Toff, D. (2018). A11y ally. `https://github.com/quittle/a11y-ally`.

Vendome, C., Solano, D., Liñán, S., and Linares-Vásquez, M. (2019). Can Everyone use my app? An Empirical Study on Accessibility in Android Apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–52.

W3C (2019). W3C Accessibility Standards Overview. `https://www.w3.org/WAI/standards-guidelines/`.

Wei, Y. (2015). MuDroid: Mutation Testing for Android Apps. Technical report, UCL-UK. Undergraduate Final Year Individual Project.

Wille, K., Dumke, R. R., and Wille, C. (2016). Measuring the accessability based on web content accessibility guidelines. In *2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 164–169.

Yan, S. and Ramachandran, P. G. (2019). The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing*, 12.

Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., and Xie, T. (2016). Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 987–992.