2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678625

Automated Repair of Size-Based Inaccessibility Issues in Mobile Applications

Ali S. Alotaibi
University of Southern California
USA
aalotaib@usc.edu

Paul T. Chiou
University of Southern California
USA
paulchio@usc.edu

William G.J. Halfond
University of Southern California
USA
halfond@usc.edu

***Abstract*—An increasing number of people are dependent on mobile devices to access data and complete essential tasks. For people with disabilities, mobile apps that violate accessibility guidelines can prevent them from carrying out these activities. Size-Based Inaccessibility is one of the top accessibility issues in mobile applications. These issues make apps difficult to use, especially for older people and people with motor disabilities. Existing accessibility related techniques are limited in terms of helping developers to resolve these issues. In this paper, we present our novel automated approach for repairing Size-Based Inaccessibility issues in mobile applications. Our empirical evaluation showed that our approach was able to successfully resolve 99% of the reported Size-Based Inaccessibility issues and received a high approval rating in a user study of the appearance of the repaired user interfaces.**

I. Introduction

Mobile devices have become one of the most essential means for accessing information and services. From e-commerce to COVID vaccine passports, the ability to access mobile applications (apps) is important for everyone. This is particularly true for the 15% of the global population with disabilities that depends on mobile devices to complete daily tasks [1], [2]. Despite the various legislation and efforts [3], [4] to ensure mobile technologies are accessible to all, mobile apps today still suffer from a wide range of accessibility issues [5], [6], [7], [8] that prevent users from interacting with the app's intended functionalities.

Touchscreen technology has been the most prominent input method for users to interact with mobile devices [9]. Yet, interacting with mobile devices by touch can be difficult for many people, such as older adults and those with motor impairments (e.g. paralysis, tremors, or neurological diseases). These difficulties can translate into imprecise touches, increased touch mistakes, or even the inability to access important functionalities in mobile apps. Studies have shown that inadequate size of touch targets is the root cause that manifests these difficulties [10], [11], [12], [13]. This type of issue, known as *Size-Based Inaccessibility Issue (SBII)* [8], occurs when the size of a touch target is less than the minimum size specified by the accessibility guidelines [14], [15], [16]. Recent studies have shown that SBIIs are among the most prevalent accessibility issues that affect mobile apps [7], [8]. In fact, a study on real-world apps from 33 app categories of the Google Play store showed that small touch target size was ranked as the second top accessibility issue [7]. Another recent large-scale empirical study on accessibility issues found that 78% of apps had more than 10% of their elements too small to pass the accessibility tests [8].

Automatically repairing SBIIs is a challenging task for several reasons. First, a repair must account for multiple SBIIs holistically in order to preserve the relative consistency of the original User Interface (UI) design. Second, due the complex relationship between Android UI components, there is no clear way of identifying the set of views and properties that needs to be modified for a given SBII. Finally, assuming that the relevant views and properties can be identified, a change in size of one touch target element can introduce further alignment or spacing issues to other areas of the UI. Together, these challenges make a seemingly simple repair difficult to achieve.

Existing approaches cannot help developers repair SBIIs. Research by Zhang et al. developed prototypes to address Android accessibility by enhancing user interactions. Their work *Interactiles* [17] focuses on making touchscreens accessible by attaching a hardware interface to the Android phone's screen to enhance tactile interaction for the visually impaired. Similar hardware overlay techniques [18], [19], [20] have been proposed in HCI research but they do not fix the underlying issues and require the hardware cutouts to be tailored to fit the devices. Software-based approaches to improve touchscreen accessibility [21], [22], [23] are more robust, but they mostly focus on using audio-based interaction techniques to allow the visually impaired to access touchscreens. Zhang also introduced "interaction proxies" to be inserted on top of an app's original UI for disabled users to more easily manipulate the app [24]. While this approach can potentially address size-based inaccessibility, it relies heavily on manually remapping interaction into new interactions. Touch Guard [25] helps users to access inaccessible small touch targets by enhancing their touched areas with screen magnification to enlarge and to disambiguate the bounds between multiple targets. These existing tools merely operate as "assistive technologies" to provide increased usability. However, they do not provide a way to help app developers repair the root causes of the problem.

In this paper, we introduce a novel approach for automatically repairing SBIIs in Android apps. Our approach defines a novel graph-based model called the Size Relation Graph (SRG) that models the visual and rendering relationships among elements in an Android UI. The SRG allows our

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00070

approach to effectively identify the set of problematic touch targets that need to be modified to repair the UI. To compute the best repair, our approach employs a multi-objective genetic algorithm to search for a solution that minimizes layout distortion. Our approach generates a repair patch and uses it to automatically generate a new APK of the repaired Android app.

The results show that our repairs can effectively fix SBIIs. In a user study that evaluated the repaired UIs, 90% of the participants rated the repaired UI as equal to or more preferred than the original. Overall, these results are very positive and indicate that our approach can help developers improve the accessibility of their mobile apps. The contributions of our paper are as follows:

1) The first-ever technique for automatically generating repairs to improve size-based accessibility in Android apps.
2) A novel graph-based model of the visual and rendering relationships among the elements in an Android UI.
3) An empirical study on real-world apps that shows our approach is effective in improving Android accessibility.
4) A user study that shows our repairs do not compromise the UI's attractiveness and are preferred for mobile usage.

Our paper is organized as follows: In Section II, we provide background information on mobile accessibility. Then in Section III we present our approach in detail, and its evaluation in Section IV. We discuss related work in Section V, and conclude in Section VI.

II. Background

An Android app consists of a set of *activities*. An *activity* is the class that creates the user interface (UI) window, and is itself comprised of a group of elements that are either *Views* or *ViewGroups*. A *View* occupies a rectangular area on the screen and is visible by default (e.g., Buttons). A *ViewGroup* is a *View* that can contain other *Views* as children. A touch target refers to any element on the UI that a user can touch, click, or interact with to perform some action. These include interactive elements (e.g., Buttons) and non-interactive elements attached to event handler that allow them to respond to user actions (e.g., implementing an $onClick$ method for an *ImageView* or a *LinearLayout*).

The visual properties of an element can be configured using a set of attributes. Android uses these attributes to determine the size and placement of elements when rendering the UI. The size of an element is expressed as a width and a height that can be configured using the attributes $layoutwidth$ and $layoutheight$, respectively. Each attribute can be specified by using a specific number (e.g., $30dp$) or using a certain size constant (i.e., '$match\_parent$' or '$wrap\_content$'). If an attribute is specified using a specific number, then it can have a size that is equal to, at most, that number (although it may not be able to achieve that size due to constraints of the UI). An element with an attribute specified as '$match\_parent$' means that the element wants to match the size of its immediate parent. An element with an attribute specified as '$wrap\_content$' means that the element wants to expand to fit its content (or children). Other attributes may also affect the size of an element. These include attributes such as $padding$, which defines the spacing between an element's borders and its encompassed content; $minWidth$ and $minHeight$, which specify a minimum bound constraint on the height and width; and $margin$, which defines the spacing between an element and its neighboring elements. Changing the value of one attribute may impact the value of the other. For example, increasing the size of an element may reduce the spacing it has with another element.

All of the elements in an *activity* can be represented in a tree-based model of the layout, called a View Hierarchy (VH). The VH contains information about the logical relationships among views (e.g., parent-child), and information about the visual aspects of each view, such as color, size, and location. Numerous tools, such as UI Automator [26], can be used to to extract an XML file representing the VH. Other tools, like Layout Inspector [27], can also parse the app's layout files to extract these attributes and then augment this information with the information obtained from dumping the state of the running UI.

Web and mobile accessibility guidelines require apps to have touch targets that are sufficient in size. The requirement is outlined in Google's Material Design principles for Android accessibility [14] and Guideline 2.5.5 of the international accessibility standard WCAG 2.1 [15]. The guidelines formally specify that mobile apps' touch targets should be at least $48dp \times 48dp$ with respect to the screen [28], [14], [16]. Testing tools such as Google Accessibility Scanner [29], Accessibility Testing Framework [30], and IBM's Mobile Accessibility Checker [31] can detect touch targets issues in mobile apps based on these guidelines. In this paper, we use the term Size-Based Inaccessibility Issue (SBII) to refer to any violations of this guideline, where a touch target falls below the required size threshold.

III. Approach

The goal of our approach is to automatically repair the SBIIs in a mobile app's UI while maintaining, as much as possible, the aesthetics and design of the original UI. Fixing SBIIs, as described in Section II, requires changes to the properties that control the size and placement of elements in the UI to allow the UI to meet the accessibility requirements. Finding the new values that fix the SBIIs while maintaining the UI's aesthetic is complicated by several challenges.

The first challenge is to ***maintain the visual consistency*** of the UI design. For example, for a navigation bar with a set of menu items, changing the size of one item without updating the other menu items will distort the navigation bar's visual consistency. The second challenge is ***knowing what needs to be changed*** in order to fix the SBIIs. Directly changing the elements with accessibility problems does not always fix the problem due to the fact that the final rendered appearance of an element depends not only on its properties, but its containing elements and nearby elements. Therefore, the set of elements and properties that need to be adjusted to fix the SBIIs often

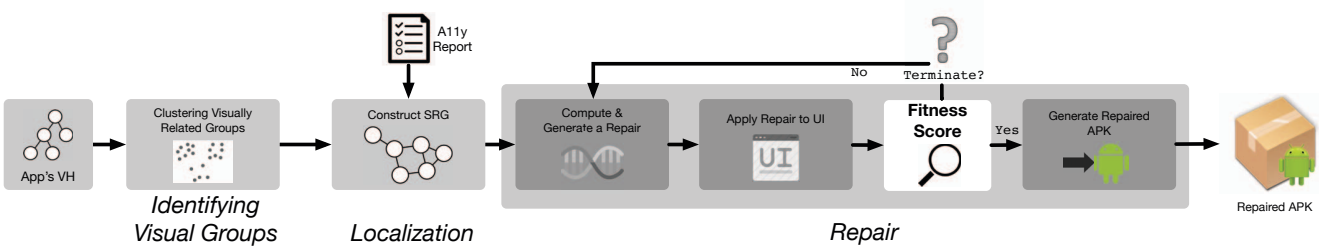

Fig. 1: An overview of our approach

include other elements in addition to the one with an SBII. The third challenge is that a repair can have a ***cascading effect***. A change to one part of a UI can trigger a chain of changes in other parts of the UI as elements change and move to accommodate the change. This challenge is compounded with the existence of multiple SBIIs in a UI or when many elements must be adjusted together to maintain the visual consistency, which increases the likelihood that the final layout will be distorted.

Figure 1 shows an overview of our approach, which can be broken into three phases, identifying visually related elements, localization, and repair. The input for our approach is an APK of an app along with a detection report that lists each of the app's Inaccessible Activities (IAs) (*activities* that exhibit SBIIs) along with details about the SBIIs of each of these *activities*. The detection report can be provided by automated detection techniques, such as Google Accessibility Scanner [29] or Accessibility Test Framework for Android (GATF) [30]

For each IA in an app, this step analyzes its layout to identify and group visually related elements that need to be adjusted together to maintain the visual consistency of the UI. The localization step then identifies and relates SBIIs to a set of elements and properties that need to be adjusted to repair the SBIIs. Finally, the repair step performs a guided search to find the best values for the identified set of elements and properties that repair the SBIIs in the IA while maintaining, as much as possible, the aesthetics of the UI. When the search terminates, the best values obtained for all selected elements are used to update the IA's corresponding static layouts. Once all of the IAs in the app are repaired, the app is compiled and provided as the output of the approach. We now explain the parts of the approach in more detail in the following subsections.

A. Phase 1: Identifying Visually Related Elements

The goal of this phase is to identify the sets of elements that should be adjusted together to maintain the visual consistency of the repaired layout. Maintaining the consistency among visually related elements (e.g., items in a menu list) is essential to maintaining the aesthetics and design of the original UI. However, identifying these sets is challenging since apps' UIs vary significantly from each other. This variation can even exist within the same app as different UIs may have their own layouts with a varying number of elements and a different set of visual relationships. This means relying on a predefined number of groups with fixed rules on how to map the elements in any UI to those groups is not practical. Instead, these groups need to be identified on a per-UI basis. A simplistic approach to identifying these groups might put elements that have the same class type (e.g., all Buttons) or style into the same group. However, in our experience this was generally inaccurate since elements with the same class type can vary widely in their appearance, and styles are not used in a disciplined way by most developers. Techniques with similar goals, but targeted to web applications (e.g., [32], [33], [34]), rely on various metrics, such as DOM structure, to group elements, and in our experience this also resulted in inaccurate groupings when applied to mobile app UIs.

To identify visually related elements, we characterized the problem as a clustering problem, where elements represent the data points that need to be made into clusters, and the cluster membership is determined by the similarity of the elements' rendering attributes. To cluster elements, our approach uses the well-known density-based clustering technique, DBSCAN [35]. This particular techniques is well suited for our problem since the algorithm (1) does not require predefining the number of clusters, and (2) produces mutually exclusive clusters (i.e., hard clustering). Both of these attributes are important for our problem domain since the variance of app UI layouts means they can have varying numbers of groupings and having non mutually exclusive clusters could prevent our search technique (Section III-C) from converging. To define the distance function, we found that (1) logical location (represented by the XPath), (2) element size, and (3) element class type consistently resulted in the most useful groupings. In our experience, elements with a similar XPaths had a higher chance of being related in terms of sharing a similar visual appearance and/or inheriting the same properties from a mutual parent (for example, icons in the navigation bar). We also found that elements with a similar size were often visually related (e.g., lists of buttons or items). Finally, we found that while element class type was, by itself, insufficient to indicate element grouping, when combined with the other dimensions, it helped to improve the grouping's accuracy.

We define our element clustering techniques as follows. First, our approach analyzes the VH of the IA and extracts each unique element and its properties. The elements become the data points that will be clustered. Next, to determine the distance between those data points, our approach defines

Fig. 2: Example that shows two apps' UIs annotated with a simplified version of the visually related groups that were identified by our clustering algorithm.

a function based on the three above-mentioned metrics. To calculate the location distance, our approach computes the Levenshtein distance between elements' XPath. The Levenshtein distance between two XPaths is the minimum number of XPath tags that need to be modified to change one XPath into the other. Our approach then normalizes the value of the location distance metric to a range of [0,1]. A metric value of zero indicates a complete match between the two elements, while one indicates a maximum difference. To calculate the size distance, our approach computes a metric for each of the size properties (height, width, and margins). If elements v_1 and v_2 have the same size propriety (e.g., height), then the metric value for that property is set to 0; otherwise, it is set to 1. Similarly, our approach computes a metric to calculate the element class type distance. If two elements have the same element class type, then the metric value is set to 0. Otherwise, it is set to 1. Our approach then calculates the overall distance as a weighted sum of the normalized value of each of the above three metrics. The weights of the metrics were determined empirically based on our experiments. The DBSCAN algorithm then uses this information to group the elements into different clusters. Each cluster then represents a set of visually related elements and the set of all clusters is the output of this phase. Fig. 2 shows a simplified version of the visually related groups identified for two mobile apps' UIs. Each number on the graph represents an identified group.

B. Phase 2: Localization of Elements and Properties

The goal of this phase is to identify, for each of the SBIIs, the set of elements and properties that need to be changed in order to repair the SBII. Although the reports from accessibility detection tools can be used to identify elements that exhibit SBIIs, they do not necessarily indicate which elements need to be adjusted to repair the SBIIs nor which properties should be adjusted.

There are several reasons for this limitation. First, the size of the element may be set based on an interaction of its properties with the properties of other elements that are located close by or of elements from which it inherits display constraints. For example, the size of an element that is set to fill the available space depends on its neighbors' size or an element's size may be bounded by the fixed size of another element in which it is visually contained. Second, maintaining the consistency of the repaired UI requires modifications to other elements, which may themselves have relationships with other elements that need to be modified to maintain consistency. One could address both of these challenges by making the repair phase of our approach consider modifying all elements and properties present in the IA. However, such a solution would dramatically increase the search space for a possible repair and means that the search process could take a long time to complete. Therefore, in this phase, our approach tries to identify a subset of all elements and properties in the IA that is (1) safe, in that it contains the elements and properties that when modified can repair the observed SBII and keep the UI consistent; and (2) minimal, to reduce the runtime needed to identify a successful repair.

To accomplish this goal, we define an approach based on building and analyzing a model of the visual and rendering relationships among the elements in an *activity*. Given a set of SBIIs, our approach can use this model to identify a minimal set of other elements and their properties that should be considered as candidates for the repair. To serve as this model, we introduce a new graph that we call the Size Relation Graph (SRG), which is defined by a tuple $\langle V, E, M \rangle$. A node $v \in V$ corresponds to an element in the IA. E is a set of directed edges that represent one of two relationships between elements in the IA: *consistency relationships*, which exist between elements that need to be changed together to maintain the visual consistency of the UI (i.e., elements that belong to the same visually related group, identified in Section III-A); and *size dependency relationships*, which exist between two elements if the size of one constrains, in some way, the size of the other. M is a function that maps each edge to a set of tuples of the form $\langle p, \varphi \rangle$. $p \in P$, where $P = \{height, width\}$, and φ is a ratio of the drawing values of p for the edge's nodes.

After building the SRG, our approach computes a subgraph for each visually related group that contains an SBII. The subgraph identifies the elements that will be targeted by the repair methodology in Section III-C. The edges of the subgraph and their corresponding annotations identify properties to be considered for the repair and provide information on how to propagate that repair to the other elements in the subgraph. To compute the subgraphs, our approach iterates over each visually related group that contains an SBII. For each such group g, our approach identifies the set of elements and properties that may need to be changed to resolve the SBII in g. To do this, our approach computes a subgraph of

the SRG that corresponds to the transitive closure of the graph originating from the element v_a in g, where v_a represents the view that has the SBII. If g contains more than one element with an SBII, our approach chooses, as v_a, the element that requires the largest size increase to fix its SBII. The intuition of selecting v in this way is that the largest size increase applied to this element will also likely repair the other elements that require a smaller size increase. The computed subgraphs are represented as a set of tuples, A, with each tuple of the form of $\langle i, sb \rangle$ where i represents the ID of the element $v_a \in V$ that contains an SBII, and sb represents the subgraph of the SRG computed for that node. We now describe each of the two edge types in more detail.

A *consistency edge* is created to represent the size relationships between elements within a visually related group g. The goal of this type of relationship is to ensure size changes are propagated among elements within g to maintain their visual consistency. For example, for the UI shown on the left-hand side of Fig. 2, our approach creates consistency edges between the nodes in the SRG that represent the three buttons in group ❹ to ensure that a change applied to one can be propagated to the others. To create the consistency edges for g, our approach iterates over its elements and for each pair of elements v_1 and v_2, our approach creates a consistency edge between their correspondent nodes in the SRG. The approach then creates an edge tuple to capture the relationship between each of the pair's dimensions (i.e., height and width) and calculate the ratio φ for that tuple by dividing the drawing value of the dimension for v_1 over the drawing value of the dimension for v_2. Returning to the example in Fig. 2, for the edge created between the nodes in the SRG representing the 'Sign up' and 'Privacy Policy' buttons, our approach creates a tuple that models the height relationship between them. Since the height of both buttons are $30dp$, the tuple will be initialized with the value $\langle height, 1.0 \rangle$.

A *dependency edge* is created to represent the size relationship that can exist between an element and one of its ancestors. The goal of modeling this type of relationship is to identify the set of nodes and properties that, given a change to a property p for an element v, may need to be changed to accommodate the change in v. To create the dependency edges, our approach iterates over the nodes in the VH of the IA. For each node v, our approach iterates through the set of its ancestors (i.e., containing layouts), starting from its parent. Then based on the size attributes defined for that ancestor, our approach will either create a dependency edge with that ancestor or skip it and move on to the analysis of the next ancestor. Our approach determines that based on the following three cases. First, if the size attribute is set as an exact number, then our approach marks that ancestor as the target node for the dependency edge. The reason for that is that an ancestor with a size attribute set as a fixed number does not change in response to the change in the size of v. Therefore, the size of v is dependent on this ancestor. Second, if the size attribute for the ancestor is set as $wrap\_content$, then our approach, will only mark that ancestor as the target node for the dependency edge if the size attribute for v was set as $match\_parent$. That is because this is the only case where that ancestor may need to be changed directly as v's size can not be directly increased. Third, if the ancestor size is set as $match\_parent$, then our approach skips that ancestor and moves to the next one. The reason is that an ancestor v_p with size set as $match\_parent$ follows the size of its own parent. Therefore, if v_p's parent increased, then v_p's size will increase, allowing v to change. The dependency edge is created between v and the identified ancestor in the SRG as determined by these three cases. The approach then creates an edge tuple to capture the relationship between the two nodes and calculates the ratio φ for that tuple by dividing the value of v over the value of the identified ancestor.

C. Phase 3: Repairing the Activity

The goal of this phase is to generate a repair that resolves the detected SBIIs in the IA without distorting the UI's appearance. Identifying such a repair is complicated for multiple reasons. First, a perfect repair may not exist due to UI layout constraints. Therefore, the repair may need to represent a tradeoff between resolving SBIIs and minimizing distortion. Second, the space of possible solutions that can be considered grows exponentially as more elements and properties need to be considered. Third, assessing the quality of a repair is difficult to do, since it requires knowing exactly how a set of proposed repairs would affect not only the changed elements but if they cause any cascading changes to other parts of the UI. Together, these challenges motivate the use of a search-based approach for finding a repair since these techniques can efficiently explore large solution spaces, consider tradeoffs in identifying a best solution, and use approximation functions to avoid expensive modeling operations (i.e., UI rendering impact).

The search-based technique we define follows the general approach of a genetic search algorithm. Therefore, we only give a brief overview below of the overall flow of the search and then describe the unique parts, the fitness function, problem representation, initial population, and repair generation, in more detail. In each iteration of the search, our approach evaluates the candidate repairs in the current population, using the metrics defined in Section III-C1, then performs selection, uniform crossover, and uniform random mutation. Our approach terminates the search once the maximum number of predefined generations has been reached or the approach reaches a fixed point where no improvement in the population has been observed for multiple generations.

1) Fitness Function: The goal of our fitness function is to guide our search to a solution that resolves as many SBII as possible. However, solutions that resolve SBIIs, may do so by increasing the size of touch targets dramatically and in a way that distorts the UI of an IA. Therefore, we design our fitness function to include not only metrics that guide the search to a UI with improved accessibility, but also metrics that penalize solutions that cause the resulting UI to significantly differ from the original or introduce new design problems. Based on our experiments with the automatically generated

touch target size adjustments, we identified several aspects of repairs that, when penalized, helped our approach to generate UIs that minimally altered the UI while generating repairs. These were: changes to view alignments, the relative position of views, spacing between views, and the amount of view size change. The fitness function for a candidate repair is calculated as the weighted sum of these four objectives.

Accessibility Heuristic: This metric represents the primary representation of how good a solution is with respect to improving the identified SBIIs. Ideally, this could be measured by inserting a solution into the app, and then running Google's Accessibility Scanner [29] on the modified app and calculating a new accessibility score based on its report. However, the process of running the scanner can take a significant amount of time. Therefore we utilized an approximation of the accessibility score. Our approach inserts a candidate solution into the app, then scans the rendered UI to identify the actual size of each touch target that had been reported as having an SBII and is still below the minimum threshold for accessible size. Simply using this number as the metric is insufficient, since it defines a step function that does not provide meaningful discernment powers among solutions where both results in the same number of SBII violations, but one may be closer. To convert this information into a gradient function with more useful notions of correctness, we calculate the amount of size that the touch targets would need to increase to satisfy the touch target minimum. This enables the approach to value solutions that are getting closer to a satisfying solution even if the resulting UI has not yet completely resolved the detected SBIIs.

Relative Positioning and Alignment of Views: Changes to the size of touch targets can cause changes to the relative position and alignment of elements in the UI as they move to accommodate the repaired elements' changed size. In some cases this can significantly distort the original layout of the UI. Therefore we introduce two metrics that favor solutions that result in lower amounts of change in the relative positioning and alignments of its elements with respect to the original UI. Our approach realizes these metrics using the following steps: First, our approach extracts the position of each element in the VH of the original UI and identifies the type of alignment and relative position it has with the other elements. For relative position, any two elements may have the following relationships: (1) intersection, (2) containment, (3) above, (4) below, (5) to the left of, or (6) to the right of. For alignment, any two elements may be (1) top aligned, (2) bottom aligned, (3) left aligned, or (4) right aligned. These relationships can be determined by comparing the x and y coordinates of each element's Minimum Bounding Rectangle (MBR). For example, two elements are bottom aligned if the y values of their bottom-right and bottom-left coordinates are equal. The same process is repeated for an IA after a candidate solution has been applied to it. Then the two sets of alignments and relative positions are compared. If a difference exists, then our approach computes the magnitude of the change by computing the minimum Euclidean distance between the current position of the changed element and where it would need to be located in order to restore the violated relationship. For the bottom aligned example, this would be the absolute difference between the y coordinates. Our approach sums the differences for all elements that have violated a prior alignment or relative position relationship and reports this as the metric for the candidate solution.

Minimum Spacing Between Views: Touch targets that expand in size can do so by expanding into the space between each pair of touch targets. However, doing so can have an impact on the layout of a UI and cause it to look very different from its original design. Therefore, we introduce a metric to favor solutions that do not cause the spacing between any pair of elements to become too small. To realize this metric our approach computes the distance between the MBRs of each pair of touch targets, and if the resulting space is below the minimum value required by Google's Material Design guidelines, then the solution is penalized. This allows solutions to utilize some of the space between touch targets but only penalizes them if it falls below this minimum value. This realization of the metric reflects our observations that many SBIIs could not be repaired without significant distortion without utilizing at least some of the space between touch targets.

Amount of View Size Change: A drawback of our accessibility heuristic is that it favors solutions that always increase the size of the touch targets. This can favor solutions that unnecessarily increase the size of the touch targets, which in turn increases the amount of distortion relative to the original UI. To penalize these changes, our approach defines a metric that favors solutions that minimize the amount of change in the size of the touch targets. To realize this metric, our approach compares the size of a touch target in the original UI (using the element's MBRs) and compares this to the size of the touch target in the UI produced by a candidate repair. The sum of all such changes in the element is used as the metric.

2) Solution Representation and Initial Population: Each candidate repair (chromosome) is comprised of a set S of tuples (each tuple corresponds to a gene), where each tuple is of the form $\langle i, p, v \rangle$. In this tuple i can refer to either a group (as defined in Section III-A) or an individual element; v denotes the amount of change or adjustment that the repair will make to i; and p indicates the property of i to which v will be applied and can be the height, spacing, or width. Our candidate repairs allow our approach to change an entire group (if i refers to a group) with one adjustment or an individual element. The group identified genes allow our approach to explore solutions that maintain consistency while the individual identified genes represent elements that have a size dependency relationship with the element containing the SBII.

For a given IA that contains SBIIs, our approach defines the genes that will be included in the chromosome in the following way. For each visually related group g and the subgraph identified for g in Section III-B, the approach first identifies the subset of properties (e.g., height or width) that might need to change to repair the SBIIs. These properties

can be identified based on the violation reported in the SBII detection report. For each such property p, the property defines a gene for g and a gene for each element that is connected to v_a in the subgraph via a dependency edge. For example, if height is the property that needs to be changed for g, and v_b is the node connected to v_a via a dependency edge, then our approach will create two tuples in S. The first tuple is created with i referring to the group g to which v_a belongs and $p = height$. The second tuple is created with i referring to v_b and $p = height$. Note that the value field v of each tuple is undefined at this point since this step only defines the chromosome structure.

Based on this chromosome structure, our approach then creates an initial population of size n of candidate solutions. For each of the n solutions the approaches creates a chromosome with the gene structure defined using the above process. Then the approach iterates over each gene and initializes its value field v by sampling a random value in a Gaussian distribution based on the element's value.

3) Generating a Repair: When a candidate solution is ready to be evaluated by the fitness function, our approach converts the solution to a repair that can be inserted into the IA. Given a candidate solution c, our approach performs the following steps: (1) For each gene in c, our approach propagates the change represented by the gene to all of the elements in the subgraph. (2) Then our approach again traverses the subgraphs capturing the changes in a set R of concrete repairs, each of which is represented as a tuple $\langle x_r, p_r, a_r, v_r \rangle$, where x_r is the XPath of the node in the VH that need to be changed, p_r is the property to be changed, a_r is the attribute that needs to be modified when applying the change to the layout files, and v_r is the new value for p of x_r. After generating R, our approach (3) rewrites the app's layout files and generates a new APK that can be run. These four steps are also used for generating the final and best solution identified by our approach.

In the first step, each gene in the candidate solution (c) is applied to each of the subgraphs. To do this our approach transitively traverses each outgoing dependency and consistency edge in each subgraph and for each edge, $v_r \rightarrow v_t$, traversed, our approach computes v_t's new value of p by multiplying the value assigned to p of v_r by the ratio, φ, defined by the edge tuple between v_r and v_t. Our approach then uses this new value of p to computes new values for the other size-related properties defined for v_t, such as padding and minimum size. This ensures that the ratio between these properties and p is maintained after the size change. For each property changed for v_t, our approach creates a corresponding node in R.

In the second step, our approach once again traverses the set of identified subgraphs. For each subgraph, our approach sets the values of the corresponding tuples in R with the value set for the node in the subgraph. For each node changed in the subgraph, our approach determines the value of a_r based on a predefined mapping between each property and the corresponding attribute used in Android for that property. This is a direct mapping except in two cases. First, when the value of the attribute that p is mapped to is defined as a $wrap\_content$, then instead of mapping p to that attribute, our approach maps p to its corresponding min attribute (e.g., $android : minHeight$). Second, If the value of the attribute is set as $match\_parent$, then the change of p cannot be directly applied to the attribute in v_t. Instead, this change is indirectly achieved by propagating the change, using the dependency edges, to a containing node.

In the third and final step, our approach iterates over the set of changes in R, and for each, our approach modifies the corresponding attributes in the app's layout files. The nodes in the subgraphs, VH, and the app' layout files all use the same identifier, which simplifies the mapping and matching between the representations. Note that we omit the details of this step, since this is mainly an engineering challenge and did not require the development of any new techniques or algorithms.

IV. Evaluation

To evaluate our approach, we designed experiments to answer the following research questions:

RQ1: How effective is our approach in repairing SBIIs in Android applications?

RQ2: How long does it take for our approach to generate repairs for SBIIs in Android applications?

RQ3: How does our approach impact the visual appeal of Android applications after applying the selected repair?

A. Implementation

We implemented our approach in Java as a prototype tool, **S**ize-based in**A**naccessibi**L**ity r**E**pair in **M**obile apps (SALEM). Our implementation uses Apktool [36] to disassemble APK resource files and repack the modified files into a new APK file. To collect UI information, we used UI Automator [26] and ADB [37] to dump the layout hierarchy files and capture the screenshots when running an app on an Android Emulator based on Android 8.0. For detecting the SBIIs in an app, we used Google Accessibility Scanner [29] and then filtered its output to capture SBIIs. To get the style information and build the VH for *activities*, we used a tool based on Layout Inspector [27] in addition to UI Automator. We ran our experiments with the following configurations: population size = 9, generation size = 8. We ran our approach on an AMD Ryzen 7 2700X 64-bit machine with 64GB memory, running Ubuntu Linux 18.04.4 LTS. The implementation of SALEM and subjects will be made available to the community via our project website [38].

B. Subjects

We conducted our experiments on a set of 58 *activities* from 48 real-world mobile apps gathered from a dataset of apps used in a recent large scale study on accessibility issues in mobile applications [7]. This dataset consists of 1,000 apps collected from across 33 categories in the Google Play store [39]. To select our subjects, we ran an accessibility evaluation tool on the dataset [7] and randomly selected 48

TABLE I. Results for SALEM's effectiveness in repairing SBIIs (RQ1) and its run time (RQ2).

		Original		Repaired		
	# of Touch Targets	**# of SBIIs**	**Accessibility Rate**	**# of SBIIs**	**Accessibility Rate**	**Running Time (mins)**
All	305	220	28	2	99	579
Average	5	4	26	0	99	9
Median	5	4	17	0	100	8
Max	20	17	93	1	100	19
Min	1	1	0	0	80	6

apps that contained SBIIs. We confirmed these reported SBIIs by manually verifying the size of each element reported. For each of these 48 apps, we selected the *activities* that the detection tool reported to have at least one SBII. From the list of SBIIs in each *activity*, we filtered out the ones that were part of *WebViews* or *AdViews*, which our approach does not handle, as they require modifying web content which our approach does not handle. In total, we have 220 SBIIs in 58 unique *activities* across the 48 subjects.

C. Experiment One

1) Protocol: To address RQ1 and RQ2, we ran SALEM on each of the subject's faulty *activities*. To account for the non-determinism of our approach's search technique, we repeated the experiment 10 times for each *activity* and reported the results based on the average numbers. To evaluate the effectiveness of our approach, for each *activity* we calculated its number of SBIIs and its *accessibility rate* before and after the repair. The number of SBIIs was determined based on the reports from Google Accessibility Scanner [29]. The *accessibility rate* was calculated as the ratio of the number of touch targets that are free of SBIIs over the total number of touch targets in the *activity*. This is a widely-used metric to measure and rank the accessibility of UIs in a mobile app [7], [8]. To address RQ2, we measured the time it took to run SALEM during the experiment.

2) Presentation of Results: The results for effectiveness (RQ1) and time (RQ2) are shown in Table I. The "Original" and "Repaired" columns correspond to the results before and after applying SALEM's repairs. We list the number of SBIIs and the resulting accessibility rate under "# of SBIIs" and "Accessibility Rate" for the original and repaired versions. We also calculated the total, average, median, maximum, and minimum (rounded to whole numbers) across all 58 *activities* for each of the metrics. Due to space constraints, we do not list the details of the subjects in the paper, but include them as supplementary material on our project website.

3) Discussion of Results: Overall, the results of our experiment show that SALEM was able to significantly reduce the number of SBIIs in the subject apps. Out of the total number of 220 reported SBIIs, our approach was able to completely fix 218 (99%) of them. The total *accessibility rate* across all 58 *activities* after the repair was 99%, compared to only 28% before the repair. These results indicate that our approach was effective in repairing the SBIIs and improving the accessibility of apps. We investigated the two SBIIs in two different *activities* that our approach could not repair and found they were UI elements whose size properties are defined in code. These SBIIs can only be repaired by analyses that would require analyzing and rewriting the source code, which is not handled by our approach.

The results for RQ2 shows that SALEM was able to generate repairs within a reasonable time. We analyzed the runtime breakdown of each individual step in our approach and found that our approach spent a significant $\sim$98% amount of time evaluating the candidate repairs by compiling a new APK for each repair and then running them on the emulator. This part can be further optimized by running the approach in parallel (e.g., using Amazon AWS).

D. Experiment Two

1) Protocol: To answer RQ3, we conducted a user-based study where we asked users to compare the original and repaired UIs. The goal of this evaluation was to understand, how our repairs affect the UI's visual layout from a user's perspective. The surveys presented side-by-side screenshots of the original and the repaired UIs, each calibrated to be shown in the resolution of the Nexus 6P mobile device that was used to run the experiment. This device has a display and resolution that is within the range of the most popular Android mobile screen sizes [40]. The order of the screenshots' placement was randomized and only labeled *Version 1* and *Version 2*.

Each survey was divided into two parts. For the first part, we wanted to measure the participants' general opinion of the original and repaired versions of the UIs. We presented the two versions and asked each participant to (1) rate their preference on a 5-point Likert scale; and (2) rate each UI's attractiveness on a numeric scale from 1 to 10. We also asked participants to provide a written explanation of their answers to understand the reason for their preference. For the second part of the survey, we wanted to measure the participants' opinion of the two versions after knowing about the accessibility improvements. We presented the same set of UI screenshots as the first part, but this time we highlighted the SBIIs on each screenshot in the same way as they would be shown in Google's Accessibility Scanner [29]. We also presented a short description explaining the issues and the functionalities that are activated by each touch target affected by the SBIIs. We then asked the participants to again, rate their preference between the original and the repaired on a 5-point Likert scale.

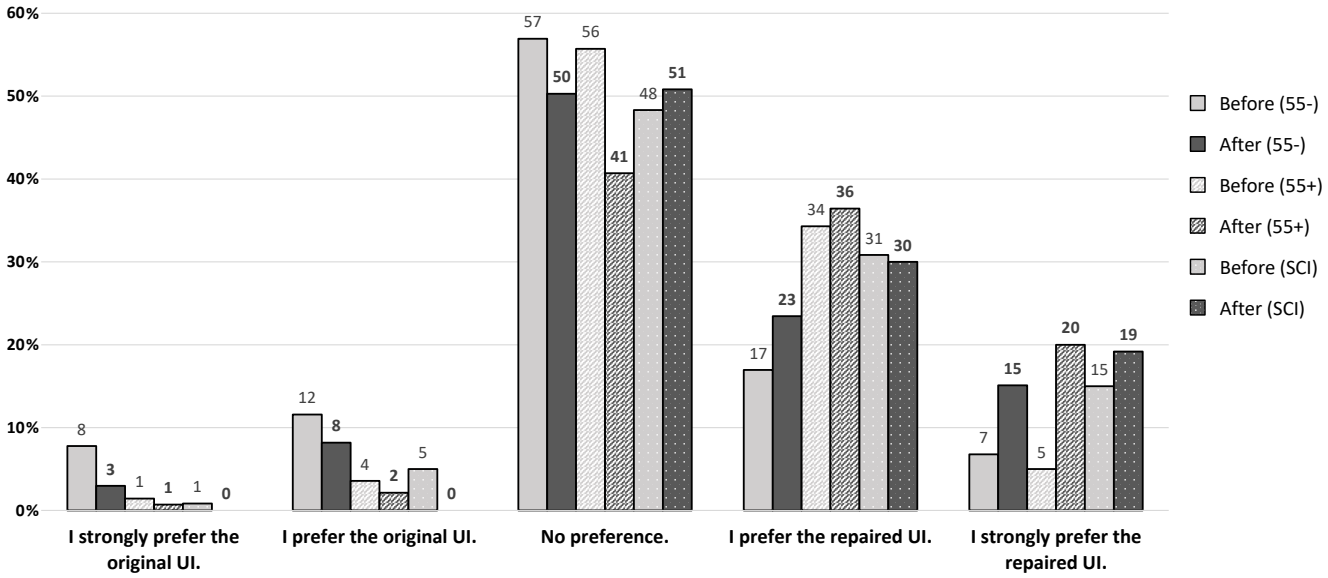

Fig. 3: Participants' preference between the original and repaired UI versions

We conducted the survey on participants from two sources: (1) Amazon Mechanical Turk (AMT), a crowd-sourcing platform that has been widely used to conduct user studies [41]; and (2) a group of disabled users that suffer paralysis with limited hand mobility. For the AMT participants, we separately collected responses from those that are under 55 years-old and those 55 years-old or older. To ensure the participants understood our instructions, we limited the locality to U.S. and Canada. We chose only those workers that had been rated as highly reliable (with an approval rating of over 98%) and who had completed over 5000 approved tasks. We also followed AMT best practices by employing a *captcha* and a check-question. In total, we had 122 responses from the 55- group, 24 responses from the 55+ group, and 20 responses from the group with motor impairment.

2) Presentation of Results: The results from the user-study are shown in Figure 3. The bar chart shows the distribution of the 5-point Likert scale preference ratings where the lighter bars are the preference ratings before accessibility awareness and the darker bars are those after accessibility awareness. We used *solid bars* to represent the group that is under 55 years-old (55- group), *striped bars* to represent the group that is 55 years-old or older (55+ group), and *dotted bars* to represent the group of users with motor impairment (SCI group).

In terms of average attractiveness, the participants rated the original (O) slightly higher than the repaired (R) with an average of (O: 6.43 R: 6.40) among the 55- group. For the 55+ and the SCI groups, the repaired version had a slightly higher rating of (O: 6.11 R: 6.25) and (O: 6.46 R: 6.94) respectively. The rating difference for the 55- group was not statistically significant (p-value = $0.57563 > 0.05$) and the rating differences for the 55+ and the SCI groups were statistically significant (p-values = $0.03327 < 0.05$, and $0.00891 < 0.05$, respectively). We used the Wilcoxon Signed-Rank test for the analysis because we were comparing paired ratings from two dependent samples and these ratings were not normally distributed.

3) Discussion of Results: The result from the user-study showed that our approach was very successful in maintaining the visual appeal of its repaired UIs. For preference, a majority of participants rated "No preference" when deciding between the original and repaired versions. This is a very good indication that our repair did not negatively affect user preference while it was able to fix almost all of the SBIIs. In fact, across all three groups, when combining "No preference" with those that prefer the repaired UI, our repair was in favor among 90% of the ratings. We investigated the comments provided by the 10% who did not prefer our repaired UI and found that the reason participants preferred the original was because they perceived smaller UI components to be more attractive. Since this is a personal preference, we do not think it undermines the quality of our repairs.

Participants preferred our repairs even more once they were aware of the implications of the SBIIs. Across the three groups, we see an average of 11% increase in favor of the repaired UI. Particularly, the number of ratings that "strongly prefer" the repaired UI doubled for the general 55- group and quadrupled for the 55+ group. This is a very strong indication that participants value accessibility and are willing to change their initial preference for the trade off. We revisited those 10% that did not favor the repair and preferred smaller layout to see whether their preference changed. Interestingly, over half of them switched to either "No preference." or preferred the repaired version, leaving only under 5% still preferring the original after awareness. The comments from the participants that switched were overwhelmingly positive, expressing that they were unaware of accessibility at first, but had no problem adapting to the repaired UI for a greater gain. One commented

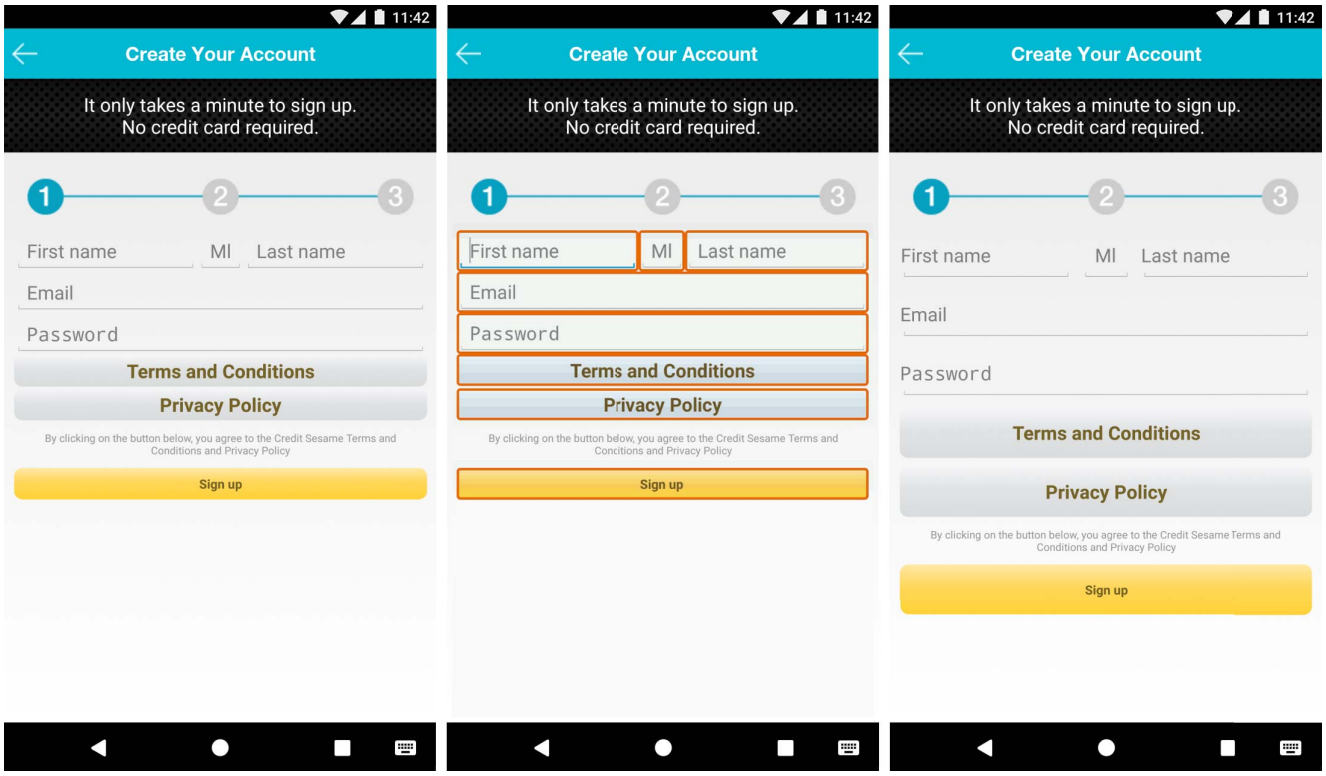

Fig. 4: Example that demonstrates SALEM. The left screenshot shows the original UI, the middle screenshot highlights the SBIIs detected by Google Accessibility Scanner, and the right screenshot shows the UI after applying SALEM's repair.

"*... and it still looks good, and now it is workable.*"

In addition to a positive impact on visual appeal, the user-study also showed that our repair was considered to be more accessible. We investigated the comments provided by the participants to understand the reason of the repaired UI being both more attractive and preferable among the 55+ and SCI groups. We found in general, these groups perceived bigger UI components to be better and more usable even before accessibility awareness. Many participants explained that the bigger touch targets from the repaired UI could help them be more efficient and avoid mistakes during interaction. Among the 19% of the SCI group that "strongly prefer" the repaired UI is a quadriplegic participant that uses his knuckles instead of fingertips to active touch screens. He explained "*The larger spacing between lines would make it considerably easier for me to access each input box with my knuckles.*" These types of insights from actual mobile users show that our approach can be impactful in addressing accessibility.

E. Threats to Validity

External Validity: The first potential threat is that the selection of participants for the user-based study in our experiment may not be representative of individuals impacted by SBIIs. To address this threat, we implemented an age question in our AMT surveys and sought motor impaired paralysis patients to ensure our participants are diverse in age and abilities.

A second threat is that the repaired UIs may not rate as well when displayed on screen dimensions different from the one we used in our evaluation. This aspect of generalizability was not tested in our evaluation. However, we believe that since our approach's focus was on maintaining *relative* visual relationships and Android uses a dynamic layout rendering approach, that repairs on screens with other dimensions would likely look similar from an aesthetics point of view.

Internal Validity: One potential threat is that screenshots used the user-based study may appear differently in size depending on the participants' displays. To mitigate this threat, we asked the participants to enter the display device they used for answering the survey and included only those results with a screen PPI that would render the screenshots to near the actual size of the Nexus 6P device's UI that was used in the emulator to generate the screenshots.

Another potential threat is that users rated the UIs based on the screenshots without directly interacting with the UIs on a mobile device. Our decision to use screenshots was for the following reasons. First, our user study does not ask users to evaluate apps' usability, which would require direct interaction with the apps. Instead, users are only asked to rate the attractiveness of the rendered UIs. Second, the use of screenshots allows us to avoid any variations in the results that may happen due to the differences in the participants' mobile devices or their selected settings. Third, the use of screenshots

allows for easy comparison as users can view the two versions of the UIs next to each other instead of having to install, run, and then uninstall different versions of our subjects. Finally, screenshots are frequently used in user-study that attempt to evaluate the attractiveness of UIs (e.g., [34], [42], [43], [44]).

Construct Validity: A potential threat is that our definition of SBIIs is dependent on the reports of GATF and the Google Accessibility Scanner [29]. The use of this definition is reasonable because it is based on Google's own Material Design research [14]. The guidelines' metric is what is considered as the accessibility threshold by experts. As further validation, we analyzed the severity of the repaired SBIIs to see how much larger they had to be in order to be considered accessible. We found that the SBIIs needed an average 56% increase in their area. Of special note, 18% of the SBIIs required doubling their touch areas and 5% of the SBIIs required an area increase of over three times to become accessible. This indicates that many of the repaired SBIIs were undersized and required significant size increases to become accessible.

Another potential threat is that the attractiveness and preference ratings by participants are subjective. To mitigate this threat, our survey is designed to measure *relative* values with either side-by-side comparison or before-and-after repair versions for the *activities*. This ensures the same pair of *activity* receives consistent ratings even though different participants may rate according to different standards.

V. Related Work

Many empirical studies in the literature have studied the prevalence of accessibility issues in mobile applications and how they impact end-users [5], [6], [7], [8]. Although they provide useful insights that drive accessibility research incentive (such as our work), they do not offer solutions to solve these underlying issues. Many Android accessibility tools today contribute to solving accessibility issues by detecting and identifying known issues based on violations of guidelines [31], [30], [7], [45], [46]. However they are only able to locate, not repair, these types of issues. Similarly, recent techniques have been developed to address various types of accessibility issues in the web domain. KAFE [47] focuses on detecting accessibility issues related to the keyboard interface. AXERAY [48] focuses on detecting semantic inconsistencies related to WAI-ARIA specifications. VizAssert [49], [50] uses formal verification techniques to detect layout-based accessibility issues in web applications.

Improving accessibility in the mobile domain has become an active area of research. Work by Wu et al. aims to promote users awareness of the built-in assistive services by automatically recommending services to users based on their needs [51]. Recent mobile accessibility research [52], [53] focuses on making mobile UI components accessible to assistive services, such as TalkBack [54], by annotating the app's interface elements with semantics and accessibility metadata. Similar work has motivated repair tools to make image-based buttons with missing content labels accessible [55]. LabelDroid [56] and COALA [57] are repair techniques based on using deep learning to automatically predict the labels of UI icons, while Brady et al. proposed a technique to suggest content labels using crowd-sourcing [58]. These approaches can repair known issues that affect disabled users from interacting with the apps via assistive technologies, but they do not address SBIIs.

Research in HCI has helped to address difficulties in touchscreen usage via both hardware and software. Hardware techniques [18], [19], [20] use physical attachments to the phone to improve tactical interactions. Software techniques provide "aids" to circumvent small touch targets by zooming or increasing size [25], [59]. While these techniques may assist disabled users on a case-by-base basis, they do not resolve the underlying root causes of SBIIs.

There has been approaches that attempt to repair general UI layout issues. OwlEye [60] focuses on repairing Android GUI layout issues, such as text overlap, blurred screen, and missing images. Other techniques, such as $\mathcal{I}$FIX [42] and $\mathcal{CB}$Repair [61], repair internationalization presentation issues in web applications, and $\mathcal{M}$FIX [34] repairs presentation issues in the mobile web environment. Although these tools target repairs of UI bugs, they do not fix SBIIs in mobile applications.

Program repair have also been the focus of many techniques from the research community [62], [63], [64], [65]. Droix [63] uses a search-based technique to automatically repair crashes in Android applications. Elixir [62] is a technique that can automatically generate patches for Java applications. The focus of this line of research is to facilitate the generation of patches to repair application crashes and defects but they do not repair accessibility issues.

VI. Conclusion

In this paper, we introduced an approach for automatically repairing SBIIs in mobile apps. Our approach builds a graph-based model of the mobile UI to identify the set of elements and properties that need to be modified to repair the UI. To identify the best repair, our approach uses a genetic algorithm guided by a fitness function that accounts for accessibility and UI distortion introduced by the repairs. Once the best repair has been found, our approach automatically generates a new APK of the repaired Android app. Our empirical evaluation showed that our approach was able to successfully resolve 99% of the SBIIs in a set of subject apps. In a user study that evaluated the repaired UIs, 90% of the participants rated the repaired UI as equal to or more preferred than the original, and valued the increased accessibility offered by the repairs. Overall, these results are very positive and indicate that our approach can help developers to improve the accessibility of their mobile apps.

Acknowledgment

This work was supported by the National Science Foundation under grant 2009045.

REFERENCES

[1] "WHO-Disability and health." [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/disability-and-health

[2] J. T. Morris, M. W. Sweatman, and M. L. Jones, "Smartphone use and activities by people with disabilities: 2015-2016 survey," *Journal on Technology Persons with Disabilities*, vol. 5, p. 50–66. [Online]. Available: http://hdl.handle.net/10211.3/190202

[3] J. Lazar, "The potential role of U.S. consumer protection laws in improving digital accessibility for people with disabilities," *U. Pa. JL & Soc. Change*, vol. 22, p. 185, 2019.

[4] "Level Access: What Accessibility Standards Apply to Mobile Applications?" https://www.levelaccess.com/what-accessibility-standards-apply-to-mobile-phone-applications/, updated: 2021-04-21.

[5] S. Yan and P. G. Ramachandran, "The Current Status of Accessibility in Mobile Apps," *ACM Transactions on Accessible Computing*, vol. 12, no. 1, pp. 1–31, Feb. 2019. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3312747.3300176

[6] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez, "Can Everyone use my app? An Empirical Study on Accessibility in Android Apps," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 41–52, iSSN: 2576-3148.

[7] A. Alshayban, I. Ahmed, and S. Malek, "Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward," 2020, p. 12.

[8] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock, "An Epidemiology-inspired Large-scale Analysis of Android App Accessibility," *ACM Transactions on Accessible Computing*, vol. 13, no. 1, pp. 4:1–4:36, Apr. 2020. [Online]. Available: http://doi.org/10.1145/3348797

[9] M. E. Mott, R.-D. Vatavu, S. K. Kane, and J. O. Wobbrock, "Smart Touch: Improving Touch Accuracy for People with Motor Impairments with Template Matching," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 1934–1946. [Online]. Available: http://doi.org/10.1145/2858036.2858390

[10] S. N. Duff, C. B. Irwin, J. L. Skye, M. E. Sesto, and D. A. Wiegmann, "The Effect of Disability and Approach on Touch Screen Performance during a Number Entry Task," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 54, no. 6, pp. 566–570, Sep. 2010, publisher: SAGE Publications Inc. [Online]. Available: https://doi.org/10.1177/154193121005400605

[11] X. Valencia, J. E. Pérez, M. Arrue, J. Abascal, C. Duarte, and L. Moreno, "Adapting the Web for People With Upper Body Motor Impairments Using Touch Screen Tablets," *Interacting with Computers*, vol. 29, no. 6, pp. 794–812, Nov. 2017. [Online]. Available: https://doi.org/10.1093/iwc/iwx013

[12] T. Guerreiro, H. Nicolau, J. Jorge, and D. Gonçalves, "Towards accessible touch interfaces," in *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*, ser. ASSETS '10. New York, NY, USA: Association for Computing Machinery, Oct. 2010, pp. 19–26. [Online]. Available: http://doi.org/10.1145/1878803.1878809

[13] L. Nurgalieva, J. J. J. Laconich, M. Baez, F. Casati, and M. Marchese, "A Systematic Literature Review of Research-Derived Touchscreen Design Guidelines for Older Adults," *IEEE Access*, vol. 7, pp. 22 035–22 058, 2019, conference Name: IEEE Access.

[14] "Material Desing Accessibility." [Online]. Available: https://material.io/design/usability/accessibility.htmllayout-and-typography

[15] "W3 Target Size." [Online]. Available: https://www.w3.org/WAI/WCAG21/Understanding/target-size

[16] "BBC Mobile Accessibility Guidlone." [Online]. Available: https://www.bbc.co.uk/accessibility/forproducts/guides/mobile/

[17] X. Zhang, T. Tran, Y. Sun, I. Culhane, S. Jain, J. Fogarty, and J. Mankoff, "Interactiles: 3d printed tactile interfaces to enhance mobile touchscreen accessibility," in *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, ser. ASSETS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 131–142. [Online]. Available: https://doi.org/10.1145/3234695.3236349

[18] B. Taylor, A. Dey, D. Siewiorek, and A. Smailagic, "Customizable 3d printed tactile maps as interactive overlays," in *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility*, ser. ASSETS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 71–79. [Online]. Available: https://doi.org/10.1145/2982142.2982167

[19] L. He, Z. Wan, L. Findlater, and J. E. Froehlich, "Tactile: A preliminary toolchain for creating accessible graphics with 3d-printed overlays and auditory annotations," in *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*, ser. ASSETS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 397–398. [Online]. Available: https://doi.org/10.1145/3132525.3134818

[20] S. K. Kane, M. R. Morris, and J. O. Wobbrock, "Touchplates: Low-cost tactile overlays for visually impaired touch screen users," in *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ser. ASSETS '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2513383.2513442

[21] S. K. Kane, M. R. Morris, A. Z. Perkins, D. Wigdor, R. E. Ladner, and J. O. Wobbrock, "Access overlays: Improving non-visual access to large touch screens for blind users," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 273–282. [Online]. Available: https://doi.org/10.1145/2047196.2047232

[22] S. K. Kane, J. P. Bigham, and J. O. Wobbrock, "Slide rule: Making mobile touch screens accessible to blind people using multi-touch interaction techniques," in *Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility*, ser. Assets '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 73–80. [Online]. Available: https://doi.org/10.1145/1414471.1414487

[23] S. Azenkot, C. L. Bennett, and R. E. Ladner, "Digitaps: Eyes-free number entry on touchscreens with minimal audio feedback," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 85–90. [Online]. Available: https://doi.org/10.1145/2501988.2502056

[24] X. Zhang, A. S. Ross, A. Caspi, J. Fogarty, and J. O. Wobbrock, "Interaction proxies for runtime repair and enhancement of mobile application accessibility," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 6024–6037. [Online]. Available: https://doi.org/10.1145/3025453.3025846

[25] Y. Zhong, A. Weber, C. Burkhardt, P. Weaver, and J. P. Bigham, "Enhancing Android accessibility for users with hand tremor by reducing fine pointing and steady tapping," in *Proceedings of the 12th International Web for All Conference*. Florence Italy: ACM, May 2015, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/2745555.2747277

[26] "UI Automator." [Online]. Available: https://developer.android.com/training/testing/ui-automator

[27] "Layout Inspector." [Online]. Available: https://developer.android.com/studio/debug/layout-inspector

[28] "Google Accessibility for Android." [Online]. Available: https://developer.android.com/guide/topics/ui/accessibility

[29] "Android Accessibility Help: Get started with Accessibility Scanner," https://support.google.com/accessibility/android/answer/6376570?hl=en, updated: 2021-04-21.

[30] "google/Accessibility-Test-Framework-for-Android," Dec. 2020, original-date: 2015-09-12T00:49:01Z. [Online]. Available: https://github.com/google/Accessibility-Test-Framework-for-Android

[31] "IBM Mobile Accessibility Checker," May 2020, original-date: 2017-11-06T14:35:17Z. [Online]. Available: https://github.com/IBMa/MAC

[32] A. Sanoja and S. Gançarski, "Block-o-Matic: A web page segmentation framework," in *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, Apr. 2014, pp. 595–600.

[33] S. Mahajan and W. G. J. Halfond, "Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2015, pp. 1–10, iSSN: 2159-4848.

[34] S. Mahajan, N. Abolhassani, P. McMinn, and W. G. J. Halfond, "Automated repair of mobile friendly problems in web pages," in *Proceedings of 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 140–150. [Online]. Available: https://dl.acm.org/doi/10.1145/3180155.3180262

[35] M. Ester, H.-P. Kriegel, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," p. 6.

[36] "Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps." [Online]. Available: https://ibotpeaches.github.io/Apktool/

[37] "Google Developers: Android Debug Bridge (adb)," https://developer.android.com/studio/command-line/adb, updated: 2021-02-18.

[38] "SALEM Project Web Site," Aug. 2021. [Online]. Available: https://sites.google.com/usc.edu/salem/

[39] Google, "Android Apps on Google Play." [Online]. Available: https://play.google.com/store/apps?hl=engl=US

[40] "Screen Sizes," https://screensiz.es/nexus-6p, updated: 2021-04-19.

[41] "Amazon Mechanical Turk." [Online]. Available: https://www.mturk.com/

[42] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond, "Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 215–226.

[43] A. Alameer, P. Chiou, and W. G. Halfond, "Efficiently repairing internationalization presentation failures by solving layout constraints," in *Proceedings of the IEEE international conference on software testing, verification, and validation (ICST)*, Apr. 2019, tex.acceptancerate: 28% (31/110) tex.pubtype: Conference.

[44] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 14:1–14:47, Sep. 2018. [Online]. Available: http://doi.org/10.1145/3241742

[45] 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 ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2014.

[46] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek, "Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, May 2021, no. 274, pp. 1–11. [Online]. Available: http://doi.org/10.1145/3411764.3445455

[47] P. T. Chiou, A. S. Alotaibi, and W. G. J. Halfond, "Detecting and localizing keyboard accessibility failures in web applications," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 855–867. [Online]. Available: http://doi.org/10.1145/3468264.3468581

[48] M. Bajammal and A. Mesbah, "Semantic Web Accessibility Testing via Hierarchical Visual Analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1610–1621, iSSN: 1558-1225.

[49] P. Panchekha, A. T. Geller, M. D. Ernst, Z. Tatlock, and S. Kamil, "Verifying that web pages have accessible layout," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–14. [Online]. Available: http://doi.org/10.1145/3192366.3192407

[50] P. Panchekha, M. D. Ernst, Z. Tatlock, and S. Kamil, "Modular verification of web page layout," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 151:1–151:26, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360577

[51] J. Wu, G. Reyes, S. C. White, X. Zhang, and J. P. Bigham, "When can accessibility help?: an exploration of accessibility feature recommendation on mobile devices," in *Proceedings of the 18th International Web for All Conference*. Ljubljana Slovenia: ACM, Apr. 2021, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3430263.3452434

[52] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu, Q. Shan, J. Nichols, J. Wu, C. Fleizach, A. Everitt, and J. P. Bigham, "Screen recognition: Creating accessibility metadata for mobile applications from pixels," 2021.

[53] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock, "Examining image-based button labeling for accessibility in android apps through large-scale analysis," in *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, ser. ASSETS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 119–130. [Online]. Available: https://doi.org/10.1145/3234695.3236364

[54] "Google: Our all-new TalkBack screen reader," https://blog.google/products/android/all-new-talkback/, updated: 2021-04-21.

[55] X. Zhang, A. S. Ross, and J. Fogarty, "Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 609–621. [Online]. Available: https://doi.org/10.1145/3242587.3242616

[56] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang, "Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning," Mar. 2020. [Online]. Available: https://arxiv.org/abs/2003.00380v2

[57] F. Mehralian, N. Salehnamadi, and S. Malek, "Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 107–118. [Online]. Available: https://doi.org/10.1145/3468264.3468604

[58] E. Brady and J. P. Bigham, "Crowdsourcing accessibility: Human-powered access technologies," *Foundations and Trends® in Human–Computer Interaction*, vol. 8, no. 4, pp. 273–372, 2015. [Online]. Available: http://dx.doi.org/10.1561/1100000050

[59] "Android Accessibility Help: Magnification," https://support.google.com/accessibility/android/answer/6006949, updated: 2021-04-21.

[60] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 398–409. [Online]. Available: https://doi.org/10.1145/3324884.3416547

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

[62] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2017, pp. 648–659.

[63] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in Android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 187–198. [Online]. Available: http://doi.org/10.1145/3180155.3180243

[64] R. S. Shariffdeen, S. H. Tan, M. Gao, and A. Roychoudhury, "Automated Patch Transplantation," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 1, pp. 6:1–6:36, Dec. 2021. [Online]. Available: http://doi.org/10.1145/3412376

[65] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, Jan. 2016, pp. 298–312. [Online]. Available: https://doi.org/10.1145/2837614.2837617