ConfFix: Repairing Configuration Compatibility Issues in Android Apps

Huaxun Huang
hhuangas@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Chi Xu
xuc2019@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Ming Wen
mwenaa@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Yepang Liu*
liuyp1@sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Shing-Chi Cheung†
scc@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

ABSTRACT

XML configuration files are widely-used to specify the user interfaces (UI) of Android apps. **C**onfiguration **c**ompatibility (CC) issues are induced owing to the inconsistent handling of such XML configuration files across different Android framework versions. CC issues can cause software crashes and inconsistent look-and-feels, severely impacting the user experience of Android apps. However, there is no universal solution to resolve CC issues and app developers need to handle CC issues case by case. Existing tools are designed based on predefined rules or visual features that are possibly manifested by CC issues. Unfortunately, they can fail or generate overfitting patches when the CC issues are beyond their capabilities. To fill the above research gaps, we first empirically studied the app developers' common strategies in patching real-world CC issues. Based on the findings, we propose ConfFix, an automatic approach to repair CC issues in Android apps. ConfFix is driven by the knowledge of how an XML element is handled inconsistently in different versions of the Android framework and generates patches to eliminate such inconsistencies. We evaluated ConfFix on a set of 77 reproducible CC issues in 13 open-source Android apps. The results show that ConfFix outperforms baselines in successfully repairing 64 CC issues with a high precision. Encouragingly, the patches for 38 CC issues have been confirmed and merged by app developers.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**.

*Yepang Liu is affiliated with both the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology.
†Shing-Chi Cheung is the corresponding author.

KEYWORDS

XML Configuration, Android, Compatibility, Automated Repair

ACM Reference Format:
Huaxun Huang, Chi Xu, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2023. ConfFix: Repairing Configuration Compatibility Issues in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598074

1 INTRODUCTION

XML configuration files are widely used in Android apps to render user interfaces (UIs) and provide essential information for execution. However, **configuration compatibility issues** (CC issues for short) can be incurred when the processing of these configuration files is inconsistent across different Android API levels, which refers to a major Android framework version (e.g., API level 28 for Android 9). Changes relating to the processing of XML configuration files are common in the Android framework [43]. CC issues induced by such changes can incur poor user experiences in Android apps, such as crashes and inconsistent look-and-feel [43]. Therefore, it is vital for developers to handle CC issues in their apps properly.

Figure 1 shows an XML element that induces CC issue (a.k.a., issue-inducing XML element) extracted from Issue #15886 of Mozilla Fenix [29], an open-source Android web browser receiving 6.4k+ stars. The app uses the `android:lineHeight` attribute in Line 4 to keep the text view's line height at `20sp`. The CC issue occurred when the app developer allowed users of older model devices to run the app at an API level earlier than 28. This is because the `android:lineHeight` attribute was newly introduced by API level 28. To fix the issue, the app developer leveraged the domain knowledge that line height is the sum of text size (`android:textSize` in Line 3 of Figure 1) and line spacing, as shown in Figure 2. It allows the use of another attribute called `android:lineSpacingExtra` in Line 5 to replace `android:lineHeight`. Therefore, the attribute `android:lineSpacingExtra` is set to `4sp` for a line height of `20sp`. By doing so, no compatibility issues occur as the attribute

```
  <TextView
      android:id="@+id/mozac_browser_tabstray_title"
      ......
      android:textSize="16sp"
-     android:lineHeight="20sp"     Not available at API level < 28.
+     android:lineSpacingExtra="4sp"
      tools:text="Firefox"/>
```

Figure 1: The code example adapted from Mozilla Fenix Issue #15886 [29].

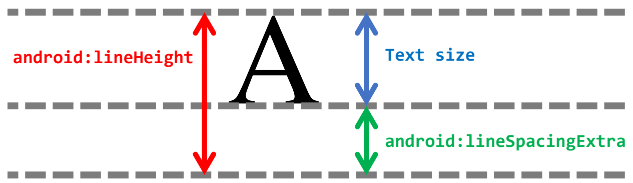


Figure 2: The process of app developers in adjusting the line height of the text view [29].

`android:lineSpacingExtra` has existed since API level 1. However, handling CC issues in Android apps needs to address two challenges.

First, the manual effort of issue detection can be non-trivial given the large number of XML elements and attributes in a typical Android app [43]. Many CC issues manifest themselves as the incorrect rendering of UIs, whose detection often involves human judgment. Diagnosing the XML elements and attribute configurations accountable for a CC issue is labor-intensive. Fortunately, the diagnosis effort can be mitigated through existing approaches (Lint [20] and ConfDroid [43]), which detect CC issues with issue-inducing elements and attributes.

Second, adapting an app to work on a set of API levels often involves resolving the CC issues arising from the evolution of the Android framework. For example, we found 1,017 usages of 21 public attributes introduced at API level 28 among 93 of 100 top-ranked Android apps. These usages are subject to CC issues at an API level below 28. However, there is no universal solution available to resolve CC issues, and app developers need to diagnose the issues and resolve them case by case. In our empirical investigation of 196 real CC issues, we observed six different patching strategies adopted by app developers (See Section 3). The issue fixing also requires domain knowledge to make the Android framework process issue-inducing elements and attributes consistently across API levels. The relevant Android API documents, such as Android Developers [6] and Android API Differences Reports [5], rarely document XML configuration changes that can induce CC issues [43]. Patches vary from case to case. The issue repair is labor-intensive, considering the intensive usage of attributes that potentially induce CC issues.

Existing approaches to repairing Android compatibility issues [38–40, 47, 62] are driven by examples of how other app developers fix compatibility issues. Such examples are either distilled manually [62] or mined from the large-scale Android app code base [38–40, 47]. However, these approaches focus on repairing issues induced by evolved Android APIs. It is non-trivial to adapt them to repair CC issues in the hierarchical-structured XML files with attribute configurations in XML elements.

In this paper, we aim at proposing an automatic approach called ConfFix to repair CC issues in Android apps. ConfFix focuses on repairing the CC issues in resource XML files that are mainly used for controlling the UI display. This is based on the observation that these are the majority of issues in the empirical dataset of real CC issues (See Section 3). Our study is based on popular open-source Android apps and reveals that developers commonly fix CC issues by modifying the attribute configuration for each issue-inducing XML element (See Section 3). However, deriving such a patch automatically is challenging due to the search space of possible candidates contributed by the number of attributes and the attribute value ranges. Take API level 33 for illustration. It supports 1,476 attributes, which can assume a value of strings, integers, or floating points. The search space is even larger if the repair needs to accommodate those patches that require changes in multiple attributes. A practical approach should be able to provide guidance in exploring such a large search space. Among existing visual-based issue repair approaches [32, 33, 44, 51–53, 59], XFix [51] is the state-of-the-art, and repairs cross-browser incompatibilities based on a predefined set of visual features, such as the layout size and positions rendered by XML elements. However, XFix is incapable of handling any issues of which inconsistent visual features are not predefined.

To tackle the above challenge, ConfFix is driven by the knowledge learned from the attribute processing logic of the Android framework among various API levels. Indeed, the Android framework supports a rich set of visual features to be configured using different XML attributes. For each issue, ConfFix first diagnoses why Android framework inconsistently processes the issue-inducing XML element across API levels. ConfFix then iteratively adjusts attribute configurations of issue-inducing XML elements to eliminate such inconsistent processing in the Android framework. By doing this, ConfFix outperforms XFix without modeling every visual feature manifested by CC issues.

We evaluated ConfFix on 77 reproducible CC issues from 13 open-source Android apps. The results show that ConfFix successfully repaired 64 issues with a high precision. To evaluate the usefulness of ConfFix, we also submitted the generated patches of previously-unknown CC issues that were successfully repaired. Encouragingly, 38 patches have been confirmed and merged.

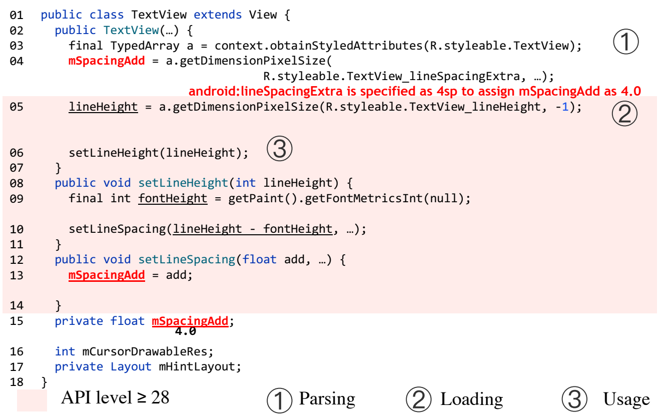
In summary, we make the following contributions:

- To the best of our knowledge, we are the first to empirically investigate real-world developers' practices in repairing CC issues.
- We propose and implement the first automatic approach ConfFix to repair CC issues in Android apps.
- We evaluated ConfFix on 77 reproducible CC issues. The results show that ConfFix significantly outperforms baselines in terms of the number of issues successfully repaired and its precision. The research artifact can be found in **https://github.com/rudmannn/ConfFix**

2 BACKGROUND & MOTIVATIONS

2.1 Configuration Compatibility Issues

XML configuration files play a key role in Android apps. There are two major types of XML configuration files in Android apps: (1) resource XML files (located in the `/res/` folder) that define apps'



```
01  public class TextView extends View {
02    public TextView(…) {
03      final TypedArray a = context.obtainStyledAttributes(R.styleable.TextView);      ①
04      mSpacingAdd = a.getDimensionPixelSize(
                                  R.styleable.TextView_lineSpacingExtra, …);
                    android:lineSpacingExtra is specified as 4sp to assign mSpacingAdd as 4.0
05      lineHeight = a.getDimensionPixelSize(R.styleable.TextView_lineHeight, -1);     ②


06      setLineHeight(lineHeight);  ③
07    }
08    public void setLineHeight(int lineHeight) {
09      final int fontHeight = getPaint().getFontMetricsInt(null);

10      setLineSpacing(lineHeight - fontHeight, …);
11    }
12    public void setLineSpacing(float add, …) {
13      mSpacingAdd = add;

14    }
15    private float mSpacingAdd;
                    4.0
16    int mCursorDrawableRes;
17    private Layout mHintLayout;
18  }
```

Figure 3: The code snippet of the Android framework on handling attributes in `TextView`.

user interfaces, and (2) the manifest file (`AndroidManifest.xml`) that are located in the root directory of the app project for defining an app's essential runtime information, such as the app's name, hardware features, and permissions. In Android apps, **XML elements** are the basic constructs within an XML configuration file (e.g., Line 1-6 of Figure 1). An XML element may contain **attributes** (e.g., `android:lineHeight`), which are defined as name-value pairs to provide information related to a specific element. The Android framework accepts attribute values in various **data types**, such as the dimension data type (`4sp` in Figure 1) to control the size of UI components. The Android framework supports configurations specified in thousands of attributes embedded in different types of XML elements. The processing of these XML elements and attributes is subject to change as the Android framework evolves, resulting in CC issues. XML elements in a configuration file are generally processed by the Android framework in three steps.

Parsing. Each XML element is parsed by `XmlPullParser`, resulting in an `AttributeSet` or `TypedArray` object that stores the attribute values in key-value pairs. The parsing process can be invoked using specific APIs in the Android framework. For example, in Line 3 of Figure 3, the API `obtainStyledAttributes` is invoked to parse the XML element in Figure 1.

Loading. The parsed attribute values can then be loaded using the *configuration APIs* to set the fields of UI objects. As an example, `getDimensionPixelSize` in Line 4 of Figure 3 is a configuration API for loading the attribute value of `android:lineSpacingExtra`. Configuration APIs are often used for assigning the parsed attribute values to the fields of UI objects at their constructors. For example, in Line 4 of Figure 3, the value of `android:lineSpacingExtra` is loaded and assigned to the field `mSpacingAdd` by the Android framework to create a `TextView` object.

Usage. The loaded attribute values are then processed to render UI objects' behavior at runtime. For example, the value of `mSpacingAdd` will flow into the API `setLineSpacing` to adjust the line spacing in the text view.

CC issues can arise if there are changes in the implementation of these three steps across API levels. As shown in Figure 3, the Android framework started to process `android:lineHeight` since API level 28. Inconsistent line heights occur when applying `android:lineHeight` at lower API levels.

Table 1: Issue patches generated by ConfFix and baselines

Method	Generated Patches
Lint [20]	-
XFix [51]	`android:height="41sp"` **(Overfitting)**
	`android:lineSpacingExtra="1sp"`
ConfFix	`android:lineSpacingExtra="1sp"`
	`app:lineHeight="20sp"`

2.2 Motivating Example

We applied two state-of-the-art techniques, Lint [20] and XFix [51], to the issue-inducing XML element in Figure 1 to illustrate the need for better automatic repair of CC issues. Lint [20] is a popular static checker released by Google. It integrates rule-based checkers that can suggest fixes when CC issues are detected. XFix [51] is a search-based technique that repairs cross-browser incompatibilities by modifying the CSS properties of HTML elements. The search process of XFix is guided by comparing the size and positions of the bounding boxes of the UI elements in the DOM tree. Although XFix was not originally designed to repair CC issues in Android apps, we adapted XFix by applying attributes documented in `TextView` to fix the issue in Figure 1. We ran XFix five times to accommodate the randomness during the search for the issue fixes. Note that we do not make a comparison with existing approaches on Android compatibility issue repair [38–40, 47, 62] since they repair the compatibility issues arising from the program code rather than the XML configuration of an app.

Table 1 shows the comparison results. Specifically, Lint cannot provide any issue-fixing suggestions because it fails to match any rules that can fix issues induced by `android:lineHeight`. XFix can generate a patch via `android:lineSpacingExtra`, the same as used in the patch made by developers. However, as the developers found, setting `android:lineSpacingExtra` to `4sp` does not achieve the same line height as `android:lineHeight="20sp"` at a higher API level. In contrast, the patch of XFix not only maintains the same line height as `android:lineHeight`, but also removes inconsistencies between API levels. Although XFix can generate a far better patch in two runs, XFix causes overfitting patches in another three runs. The reason is that the fitness function adopted by XFix does not appropriately interpret the semantic of `android:lineHeight` by considering only the layout bounds rendered in a DOM tree. As such, `android:height` is suggested as a repair candidate to achieve consistent layout bounds across API levels.

Our proposed approach, ConfFix, overcomes these limitations by consistently generating the correct patches among five runs. Besides `android:lineSpacingExtra`, ConfFix generates another patch by using `app:lineHeight`, which is located in the AndroidX library that Android officially releases for resolving incompatibilities. The rationale of ConfFix is to monitor how the attributes are handled in the Android framework. Specifically, ConfFix aims to eliminate inconsistent field values induced by issue-inducing attributes among

API levels. Figure 3 shows the code changes that induce CC issues in Figure 1. In Line 5, the configuration API `getDimensionPixelSize` is invoked to load `android:lineHeight`, whose attribute value will be stored in the variable `lineHeight` as `20.0`. Such a variable is further processed by the Android framework to adjust the extra space after each line (`mSpacingAdd=4.0` in Line 13) while considering the text view's font height (`fontHeight=16.0` in Line 9). Since `android:lineHeight` was introduced at API level 28, it cannot be used to assign a value to `mSpacingAdd` at low API levels, causing the CC issue shown in Figure 1. Therefore, the app developers specified `android:lineSpacingExtra`, which was introduced at API level 1, as `1sp` to make the assignment of `mSpacingAdd` the same as when using `android:lineHeight`. The attribute `app:lineHeight` also follows the above principle to fix the issue.

The above example demonstrates how ConfFix performs issue repair by diagnosing the inconsistent text view's line height as a result of the field `mSpacingAdd` in the Android framework. Given the large search space of patches, ConfFix is guided to search for issue-fixing attributes (e.g., `android:lineSpacingExtra`) that can dynamically affect `mSpacingAdd`. ConfFix further adjusts the issue-fixing attribute values to assign consistent values to fields across API levels. Such a design fills the gap of XFix, which fails to model visual features relevant to CC issues (e.g., text view's line height).

We need to tackle the following technical challenges to achieve the above research goals. First, how to precisely diagnose the fields related to CC issues in the Android framework. Second, how to generate patches to eliminate such inconsistent field values induced by CC issues. To further provide insights into the design of ConfFix, we conducted an empirical study on the common strategies of app developers for repairing CC issues.

3 PATCHING STRATEGIES OF DEVELOPERS

To empirically study common patching strategies for CC issues, we refer to the state-of-the-art dataset released by Huang et al. [43] in November 2021, which contains code revisions of 196 CC issues from 43 open-source Android apps. Specifically, 190 issues (190/196=96.9%) are related to the resource XML elements used to control apps' UI display (See Section 2). The above finding motivates us to **design an approach to fix CC issues in resource XML elements**. We discarded six CC issues located in the manifest files (i.e., `AndroidManifest.xml`).

We performed the data analysis as follows. We first randomly sampled 95 issues (50%) from the empirical dataset. For each issue, two authors with two years of experience in Android app development independently analyzed the code revisions and related issue reports to identify code snippets related to patches. A pilot taxonomy was constructed by gathering the results of two authors and resolving conflicts during meetings. Then, the two authors iteratively labeled the remaining 95 issues and held discussions about adjusting the pilot taxonomy and resolving conflicts. The final results were obtained once the two authors reached a consensus on the taxonomy and the labels of the empirical dataset.

We successfully tracked patches for 150 CC issues. The app developer only flagged the presence of the remaining 40 CC issues in their code revisions without tackling the resulting inconsistent runtime behavior across API levels (e.g., supporting new features

drawable/divider_dark.xml
```
01    <shape xmlns:android="http://schemas.android.com/apk/res/android">
02 -      <solid android:color="?attr/list_separator_color" />
03 +      <solid android:color="@color/list_separator_dark" />
04    </shape>
```
drawable/divider_light.xml
```
05    <shape xmlns:android="http://schemas.android.com/apk/res/android">
06 -      <solid android:color="?attr/list_separator_color" />
07 +      <solid android:color="@color/list_separator_light" />
08    </shape>
```
styles-dark.xml
```
09    <item name="list_separator_drawable">@drawable/divider_dark</item>
```
styles-light.xml
```
10    <item name="list_separator_drawable">@drawable/divider_light</item>
```
ListCardView.java
```
11 +  ResourceUtil.getThemedAttributeId(getContext(),
                       R.attr.list_separator_drawable), true));
```

Figure 4: The patch in apps-android-wikipedia 4738471 [7]

introduced at higher API levels). The patches for these 40 issues cannot be tracked from the related issue reports. Overall, most issues were fixed at the granularity of attributes (101/150=67.3%), meaning that developers fixed issues by changing the attribute configurations of the issue-inducing XML elements. The other issues were fixed at the granularity of XML elements (49/150=32.7%), meaning that developers found alternatives to replace the issue-inducing XML elements. Note that it is possible for app developers to combine different patching strategies to fix a CC issue. For example, the issue in Figure 4 was fixed by combining the T2 and T3 strategies as illustrated below. We found five such cases in our empirical dataset.

3.1 Patches at the Attribute Granularity

We found 101 issues whose patches are at the attribute granularity. Four fixing strategies, referred to as T1–4 below, were found for these 101 issues.

T1: Specifying Issue-Fixing Attributes. We observed 55 issues were fixed by specifying issue-fixing attributes in the issue-inducing XML elements to eliminate the inconsistent runtime behavior. For example, to fix the issue in Figure 1, the issue-fixing attribute `android:lineSpacingExtra` was specified to adjust the text view' line height for `android:lineHeight`, which is not available at API levels < 28.

T2: Changing Values of Issue-Inducing Attributes. Twenty-six issues were fixed by directly changing the values of issue-inducing attributes. This usually happened when the issues were caused by attribute data type changes between different API levels. As shown in Figure 4, project apps-android-wikipedia's commit 4738471 [7] files such an issue, which is induced by `android:color` not accepting the style data type (i.e., `list_separator_color`) at API levels < 21. To resolve this issue, developers separated `list_separator_color` as two different color data type values (Line 3 and 7 of Figure 4).

T3: Invoking APIs in App Code. We observed 20 issues that were fixed by invoking APIs in the app code to simulate the runtime behavior of issue-inducing attributes. As Figure 4 shows, the app developers invoked the issue-fixing API `getThemedAttributeId` to enable the separated color data type values for different styles at lower API levels.

mipmap-anydpi-v26/icon_round.xml

```
<adaptive-icon
    <background android:drawable="@color/primary"/>
    <foreground android:drawable="@mipmap/icon_foreground"/>
</adaptive-icon>
```

<adaptive-icon> not available for API level < 26

mipmap-hdpi/icon_round.png

Figure 5: The patch in AdAway 576720b [1].

```
- <ripple android:color="@color/safr">
-     <item android:drawable="@android:color/white"/>
- </ripple>
```

<ripple> not available for API level < 23

```
+ <selector xmlns:android="http://schemas.android.com/apk/res/android">
+    <item android:state_pressed="true"
+          android:drawable="@color/safr_pressed" />
+    <item android:drawable="@android:color/white" />
+ </selector>
```

Figure 6: The patch in AmazeFileManager b3b8d60 [4].

T4: Directly Removing Issue-Inducing Attributes. We observed five cases where app developers directly removed issue-inducing attributes. For example, apps-android-wikipedia's commit 53010ec [8] files an issue due to `android:drawableLeft` co-exists with `android:drawableStart` at API levels < 23. The issue was fixed by simply removing the redundant attribute `android:drawableLeft`.

3.2 Patches at the XML Element Granularity

The remaining 49 issues were fixed by using alternatives to replace the issue-inducing XML elements. This usually happens when such issue-inducing XML elements are not available at low API levels. We found two common strategies, referred to as T5 and T6 below.

T5: Specifying Non-XML Resource Files. We found 38 issues fixed by specifying non-XML resource files that achieve the same visual appearance as issue-inducing XML elements. As shown in Figure 5, AdAway's commit 576720b [1] files an issue induced by using `<adaptive-icon>` to specify the app's desktop icon. Such an `<adaptive-icon>` tag is not available for API levels < 26. Therefore, the app developers introduced a set of PNG files that manifest the same visual appearance as configured by `<adaptive-icon>`.

T6: Specifying Different Types of XML Elements. Another 11 issues were fixed by specifying different types of XML elements. As shown in Figure 6, AmazeFileManager commit b3b8d60 [4] files an issue induced by `<ripple>` not available at API levels < 23. The app developers fixed this by the XML elements with the `<selector>` tag instead, to manifest the ripple effect at lower API levels.

4 CONFFIX APPROACH

We propose ConfFix as a guided search-based approach to repair CC issues in Android apps. As illustrated in Section 3, ConfFix focuses on repairing CC issues in resource XML files, which are often configured to render the UI display of Android apps. ConfFix edits the attribute configurations of issue-inducing XML elements by either **finding appropriate issue-fixing attributes and attribute values** (T1, T2 in Section 3) or **removing the issue-inducing attributes** (T4 in Section 3) for the issue-inducing XML element. Such patching strategies account for 57.3% (80/150) in our empirical dataset. ConfFix does not support T3, considering the small proportion (20/101) and the challenge, which requires determining (1) where to edit the app code, (2) which APIs to select, and (3) how to apply context information for valid API calls. We leave the adoption of T5 and T6 as future works since they are at the XML element granularity and thus require different strategies compared to the patches at the attribute granularity. Specifically, applying T5 requires image files for issue-inducing elements. Applying T6 requires issue-fixing XML elements, which is more challenging considering the larger search space of multiple XML elements organized in a hierarchical structure. Existing approaches are ineffective in modeling possible Android visual features that can be manifested by CC issues. The basic idea of ConfFix is to first diagnose how the inconsistent handling of issue-inducing attributes across API levels can affect the apps' UI display, and then search for a patch to eliminate such inconsistencies. As described in Section 2, the handling of XML attributes in the Android framework involves a set of fields that are critical for rendering the apps' UI (e.g., `mSpacingAdd` in Figure 3 for handling `android:lineHeight`). ConfFix generates patches to eliminate the differences in these field values across different API levels while preserving the semantics as expected by app developers. As the example in Section 2 shows, the patches produced by ConfFix not only eliminate the inconsistencies in the field `mSpacingAdd` between API levels 27 and 28 but also preserve the `20sp` line height as expected by app developers.

ConfFix works as follows. Given a subject app app_i, a CC issue i, and the UI test script U that can reproduce i, ConfFix outputs a patched app app_p. Formally, a CC issue i is defined as a tuple $\langle e_i, A_i, l_i \rangle$, where A_i is a set of issue-inducing attributes located in the issue-inducing XML element e_i. Whereas, l_i is the API level where inconsistent UI rendering is induced by comparing A_i to the target API level l_t. ConfFix performs issue repair in two stages. In the first stage, after replaying the UI test script U in app_i, ConfFix performs **issue diagnosis** by identifying a set of fields F that are affected by A_i and have inconsistent values between API levels l_i and l_t. For ease of presentation, we refer to the fields in F as **key fields**. Such a set of key fields F is leveraged to calculate the fitness score that quantifies how well the patches generated by ConfFix preserve the semantics as app_i at API level l_t. In the second stage, ConfFix **generates patches** by editing the attribute configuration of issue-inducing XML elements to adjust the values of F at API level l_t. ConfFix outputs app_p that reaches the maximum fitness score without inconsistent values in F between API levels l_i and l_t.

4.1 Issue Diagnosis

In this step, ConfFix diagnoses the root causes of the issues in e_i after replaying the UI test script U in app_i. Essentially, ConfFix identifies the set of key fields F that hold inconsistent values between the two API levels due to the issue-inducing attributes A_i. As discussed in Section 2, the key fields in F are located in the objects O_i created due to the processing of e_i. The `TextView` instance in Figure 3 is an example of O_i created from the XML element in Figure 1. We refer to the objects in O_i as **key objects** for ease of presentation. The diagnosis process involves two steps: (1) identifying the key objects O_i of e_i, and (2) identifying the key fields F for each object in O_i.

4.1.1 Key Object Identification. In this step, ConfFix identifies a set of key objects O_i created due to the processing of e_i. Specifically, ConfFix leverages the mechanism documented in the Android Developers [9] and identifies key objects for two main cases.

The key objects O_i can be accessed through resource IDs, which are unique resource names for XML elements. Specifically, ConfFix inserts code snippets into the app source code to identify key objects. First, ConfFix finds the activities or fragments that renders e_i. ConfFix inserts `findViewById(id)` with the resource id as the parameter into the callbacks responsible for rendering views (e.g., `onCreate` in Activity, `onViewCreated` in Fragment). The return values of inserted `findViewById(id)` are identified as O_i. If e_i is an element whose key objects cannot be identified from the previous step, ConfFix tracks the API invocations that take the resource ID of e_i as a parameter and return the resource object in the app code. ConfFix identifies the objects returned from these API invocations to O_i as they can be potentially related to e_i.

Although we cannot draw a whole picture of how to identify objects created from XML elements, adopting the above two strategies does not significantly impact the CC issue repair. In our evaluation, we found that ConfFix failed to identify key objects for two out of 77 CC issues.

4.1.2 Key Field Identification. After identifying key objects O_i, ConfFix continues to identify a set of key fields F holding inconsistent values between the two API levels due to the issue-inducing attributes A_i. ConfFix will further compare the values of such key fields F as the fitness score for patch generation.

However, identifying such key fields requires comparing field values to check whether there are any inconsistencies. Such a task is non-trivial as classes in the Android framework may contain a set of non-primitive fields. Many non-primitive fields in the Android framework are incomparable. For example, the field `mHintLayout` in Line 17 of Figure 3 is an instance of the `Layout` class, which is incomparable without overriding the `equals` or `compareTo` APIs. We found that among the 13,504 classes in the Android framework at API level 33, only 768 classes override the `equals` API, and 17 classes override the `compareTo` API.

ConfFix facilitates the comparison of non-primitive fields by measuring the differences of a set of primitive fields accessible from O_i. We follow the definition of Tripp et al. [55] to model field accesses as access paths. An access path of the key object $o_i \in O_i$ is represented in the form of $o_i.g.h.f_p$, where f_p is the primitive field that is reachable by o_i, and g and h are intermediate fields in non-primitive types for o_i to reach f_p. We define the length of access path as the number of fields involved from o_i to reach f_p (e.g., length 3 for $o_i.g.h.f_p$). ConfFix collects access paths up to a length k, otherwise the number of collected access paths becomes infeasible large [34]. Note that there is no general value of k available to fit all cases. For example, $k = 1$ is appropriate for the issue in Figure 1 to observe the inconsistencies in `mSpacingAdd` between API levels 27 and 28. ConfFix dynamically adjusts k for each individual case.

Algorithm 1 shows how ConfFix identifies key fields F from $o_i \in O_i$. In particular, ConfFix generates a length-k access path

Algorithm 1: Key Field Identification

```
  Input: $O_i$: The key objects created from $e_i$
  Output: $F$: The set of access paths collected from $e$
1 foreach $k \leftarrow 1$ to $k_{max}$ do
2     $AP \leftarrow accessPathsOfPrimitiveFields(k, O_i)$
3     $F \leftarrow inconsistentFieldValues(AP, l_i, l_t)$
4     if $len(F) > 0$ then
5         return $F$
```

$o_i.f_1.\cdots.f_{k-1}.f_k$ by expanding the field f_k located in the non-primitive field f_{k-1} (Line 2). ConfFix then identifies key fields F whose values are affected by A_i and demonstrate inconsistencies between API levels l_i and l_t (Line 3). Specifically, ConfFix generates app_i' by removing A_i from e_i, and then considers a field f is affected by A_i if its values are inconsistent between app_i and app_i' at API level l_t. The value of k increases when ConfFix fails to identify valid key fields at the current k value. The algorithm terminates when no key fields are extracted and k reaches the maximum number k_{max} (set to 5 by default). The above process ignores key fields whose lengths are greater than k while being able to guarantee the performance of ConfFix. In our experiments on 77 real CC issues, we did not observe ConfFix failing or generating overfitting patches for this reason.

4.2 Fitness Score Calculation

Given a plausible patch app_{pls} (the patch that passes U and shows consistent values of F between l_i and l_t), the fitness score $Score$ is calculated to measure how close the values of F in app_{pls} are to app_i at l_t, as shown in Equation 1.

$$Score(app_{pls}) = 1 - \frac{FDiff(app_{pls}, app_i)}{FDiff(app_i', app_i)} \tag{1}$$

where the function $FDiff(app, app')$ measures the differences in the values of F between two apps app and app' at API level l_t. Intuitively, $Score$ measures how well app_{pls} can eliminate the differences in the values of F at API level l_t before and after removing the issue-inducing attributes A_i in e_i. ConfFix calculates $Score$ for app_{pls} after replaying U. Initially, $Score = 0$, since no differences in key field values are eliminated. ConfFix outputs a patched app app_p with the maximal $Score$ value.

$FDiff(app, app')$ is calculated as the sum of $\Delta(f, app, app')$, which measures the differences in terms of the value of each $f \in F$ between app and app' at API level l_t, as shown below.

$$FDiff(app, app') = \sum_{f \in F} \Delta(f, app, app') \tag{2}$$

However, there is no generic way to calculate $\Delta(f, app, app')$, as many primitive fields in the Android framework are numerically incomparable. The field `mCursorDrawableRes` in Line 16 of Figure 2 is such an example of the text cursor's resource ID being stored. On the other hand, simply checking all the key fields for equality via $\Delta(f, app, app')$ fails to capture the subtle changes made during the patch generation process. As illustrated in Section 2, the numerical differences of `mSpacingAdd` are helpful to search appropriate values for `android:lineSpacingExtra` and `app:lineHeight`. Identifying numerically comparable fields is non-trivial without explicit labels provided in the Android framework.

We make the following design choice. For an A_i that contains a_i in the numeric data type (i.e., a_i in integer, float, dimension and fraction data types, as documented in Android Developers [6]), the numerical differences in the key fields are essential in the search for an appropriate numerical attribute value for a_i. In this case, ConfFix calculates $\Delta(f, app, app')$ for $f \in F$ in the numeric primitive data type as shown in Equation 3.

$$\Delta(f, app, app') = |v(f, app) - v(f, app')| \tag{3}$$

where $v(f, app)$ and $v(f, app')$ are the values of f in app and app' at API level l_t. For the other cases, ConfFix simply checks the equality of f, as Equation 4 shows.

$$\Delta(f, app, app') = \begin{cases} 1, & v(f, app) \neq v(f, app') \\ 0, & v(f, app) = v(f, app') \end{cases} \tag{4}$$

It shows threats that ConfFix may fail or generate overfitting patches by comparing numerical differences for those incomparable primitive fields (e.g., `mCursorDrawableRes`). However, we have not witnessed such cases in our evaluation of 77 real-world CC issues.

4.3 Patch Generation

Guided by the fitness score, ConfFix generates patches by editing the attribute configuration of e_i. Specifically, ConfFix first identifies candidate issue-fixing attributes A_f from the Android framework. Then, ConfFix seeks a set of candidate patches APP_{cand} by searching for an attribute value that achieves an optimal fitness score for each $a_f \in A_f$. Finally, ConfFix seeks the best combination APP^*_{comb} of candidate patches in APP_{cand} to facilitate issue repair via multiple attributes. The patched app app_p is generated by applying APP^*_{comb} to e_i.

4.3.1 Search for Issue-Fixing Attributes. In this step, ConfFix identifies a set of candidate issue-fixing attributes A_f in the Android framework. The above task is non-trivial considering the large number of public attributes in the Android framework (1,476 at API level 33). Based on the empirical findings of developers' practices in choosing issue-fixing attributes (83 issue-fixing attributes in 55 T1 issues), ConfFix identifies A_f as follows.

- In the Android framework class cls_i that processes e_i at API level l_i, ConfFix considers all attributes with the same data types as candidate issue-fixing attributes, and puts them into A_f. This heuristic is based on our finding on the patches of 27 T1 issues.
- ConfFix further searches for candidate issue-fixing attributes from the AndroidX library. Specifically, ConfFix first searches for the attributes in the AndroidX library with the same name as a_i as A_f. If the above step fails, ConfFix searches the AndroidX library for a class that extends from cls_i, and adds all attributes that are used in this class and with the same data type as a_i to A_f. This heuristic is based on our findings on the patches of 11 T1 issues.

Table 2: The process of ConfFix in generating a candidate patch by `android:lineSpacingExtra` for the issue in Figure 1.

Run	1	2	3	4
Attr. Value	`20sp`	`16sp`	`0sp`	`1sp`
Score	< 0.0	< 0.0	0.0	1.0

- If ConfFix does not find A_f from the above steps, it considers all attributes in cls_i as candidate issue-fixing attributes, and puts them to A_f.

By doing so, the search space for issue-fixing attributes is significantly reduced without trying all possible attributes in the Android framework. In our evaluation of 77 real CC issues, ConfFix can successfully fix 64 of them by adopting the above strategies.

4.3.2 Search for Candidate Patches. For each candidate issue-fixing attribute $a_f \in A_f$, ConfFix generates $app_{cand}(a_f)$ by searching for an attribute value that maximizes *Score*. Specifically, for a_f that is numerically incomparable (e.g., `android:color`), ConfFix iterates all attribute values with the same data type in e_i and selects the one with the highest *Score*. For a_f that are numerically comparable (e.g., `android:lineSpacingExtra`), ConfFix searches for the attribute value inspired by Mahajan et al [51]. Specifically, ConfFix first selects the attribute value in e_i that has the same data type and reaches the highest *Score* value as the starting point for adjustment. ConfFix uses the default value of a_f as the starting point if there is no improvement *Score* in the previous step (i.e., $Score < 0$). ConfFix then adds a random delta value to a small value range (i.e., $(0, d]$) of its attribute value from the starting point. Initially, d is set to 1. ConfFix expands d exponentially if *Score* is improved. If *Score* is not improved, ConfFix switches to the opposite direction (i.e., $[-d, 0)$ value range) from the current value and resets d as 1. A candidate patch is generated if there is no fitness score improvement in any of the exploratory directions (i.e., local optima).

Table 2 shows how ConfFix generated a candidate patch for the issue in Figure 1 by modifying `android:lineSpacingExtra`. To search for an appropriate starting point, ConfFix first explored the dimensional attribute values specified in e_i (i.e., `20sp` and `16sp`) and obtained negative *Score* values (Runs 1 and 2). ConfFix then selected `0sp`, the default value of `android:lineSpacingExtra`, as the starting point and incrementally updated the attribute value (Run 3). The exploration terminated when the attribute value was `1sp` with 1.0 fitness score (Run 4).

4.3.3 Search for Candidate Patch Combination. ConfFix seeks the best combination of candidate patches APP^*_{comb}. This process is included as a patch involving one issue-fixing attribute may only partially resolve a CC issue and should be combined with other issue-fixing attributes. We found that 26 CC issues in our empirical study dataset were fixed by multiple issue-fixing attributes. Inspired by the process proposed by Mahajan et al. [51], ConfFix searches for APP^*_{comb} in a biased random fashion according to the fitness score of each candidate patch $app_{cand} \in APP_{cand}$. Specifically, app_{cand} is added to a combination APP_{comb} with the probability as follows:

$$P(app_{cand}) = \frac{Score(app_{cand})}{Score_{max}(APP_{cand})} \tag{5}$$

where $Score(app_{cand})$ represents the fitness score of app_{cand}, and $Score_{max}(APP_{cand})$ represents the maximum fitness score of the candidate patches in APP_{cand}. ConfFix applies APP_{comb} to e_i and computes $Score(APP_{comb})$ following the process in Section 4.2. ConfFix preserves the candidate patch combination APP^*_{comb} with the highest fitness score. A patched app APP_p is generated by applying APP^*_{comb} to e_i.

The algorithm terminates when (1) $Score = 1.0$ (i.e., achieving consistent values in F as app_i at API level l_t) and no inconsistencies observed in F between API levels l_i and l_t; (2) the algorithm reaches a maximum threshold of candidate patch combinations (20 by default); (3) the algorithm achieves no improvement in fitness score by generating a sequence of candidate patch combinations (5 by default); or (4) the algorithm exceeds the time budget.

5 EVALUATION

We implemented ConfFix based on UIAutomator2 [30], an open-source UI testing framework for Android apps. We evaluate ConfFix by answering the following research questions:

- **RQ1: (Effectiveness)** How effective is ConfFix in repairing CC issues in real-world Android apps?
- **RQ2: (Comparison)** Can ConfFix outperform existing approaches in repairing CC issues?
- **RQ3: (Usefulness)** Are the patches generated by ConfFix useful to app developers?

5.1 Evaluation Setup

5.1.1 Issue Dataset Collection. To answer the above RQs, we collected a set of 77 real-world CC issues by the following process.

To measure the quality of patches generated by ConfFix and baselines, we collected a set of reproducible CC issues that **have been fixed** by app developers in the past (i.e., issues with human patches). We collected issues from Android app projects with rich project maintenance history. Specifically, we used "android" as the keyword to search for open-source app projects on GitHub [18]. From the returned projects of more than 1,000 stars, we collected 118 projects that contain APK files in the project release or the "app" keyword in the project name, as the projects satisfying these conditions are more likely to be Android apps. We further excluded the projects meeting any of the following conditions: (1) the project is not an Android app (e.g., a third-party library); (2) the project contains less than 1,000 code revisions; and (3) the project is included in the empirical dataset [43]. In total, 38 app projects were selected. We collected code revisions related to CC issue repair following the keyword-based search process in [43]. For a balanced collection of CC issues among app subjects, we kept the latest 200 code revisions returned by keyword-based searches for each app subject. In total, 3,089 code revisions in 38 app projects were collected. Two authors performed issue reproduction by (1) screening out code revisions not related to CC issues as those code revisions can be accidentally included by our keyword-based search; (2) identifying the buggy app versions; (3) identifying the issue-inducing attributes and XML elements from the revision-related commit logs, issue reports, and code diffs; and (4) building UI test scripts in Android emulators to manifest inconsistent runtime behavior induced by the CC issues. A code revision was excluded if (1) we failed to recover the issue reproduction steps by referring to information disclosed in the app project (e.g., issue discussions, app user manuals); (2) its manifestation requires special triggering conditions (e.g., interaction with other accounts); and (3) the issue occurs at API levels < 21 as their market share is less than 1% [28]. Eventually, we collected 35 issues.

We further expanded the issue dataset with reproducible CC issues that **have not yet been fixed** to seek for developers' feedback on the patches generated by ConfFix. From 4,254 apps in F-Droid [15], we sampled 537 apps that satisfy the following conditions: (1) the app project received 30+ stars on GitHub [18]; and (2) the last git commit was pushed into the most recent month. We ran the rules of ConfDroid, the state-of-the-art approach for CC issue detection, to detect CC issues at API levels ≥ 21 for these 537 Android apps. In total, ConfDroid generated 706 warnings in 110 apps. Due to the cost of the issue reproduction process, we sampled 25% of apps that were recently updated and not included in the empirical dataset (194 warnings in 27 apps). Two authors reproduced the warnings following the process of [43]. Specifically, we tried to reach the `Activity` where the XML element is located. By reading the code of the `Activity`, we tried to manifest inconsistent runtime behavior induced by A_i. Eventually, we collected 42 issues. The remaining warnings failed because (1) we cannot find clues in the app code to reach the XML element; (2) the CC issues can only be triggered under special conditions; and (3) ConfDroid generates false warnings.

Table 3 shows the statistics of all 77 (35 + 42) CC issues, which cover 10 distinct issue-inducing attributes. These issues come from 13 Android apps that are diverse (concerning multiple app categories in Google Play [19]) and popular (with thousands to millions of downloads in Google Play [19]). Note that the number of CC issues collected from each app subject varies depending on the usage of issue-inducing attributes. For example, ConfDroid generated 105 warnings of `android:drawableTint` in FairEmail [23]. We successfully reproduced 32 warnings but failed to reach the remaining XML elements following the above process.

5.1.2 Baselines. We compared ConfFix with the following two baselines.

- XFix [51], the state-of-the-art tool for repairing cross-browser inconsistencies. We leveraged XFix's fitness function, which quantifies the relative layout deviation of UI elements, to fix CC issues. Note that the original version of XFix generates patches by selecting appropriate CSS properties from a manually-maintained list. For CC issues, given an issue-inducing XML element e_i, we adapted XFix to apply the attributes of e_i. We kept the other original behavior because the additional inputs required by XFix (DOM tree, screen images) are available in the Android emulator.
- Lint [20], a static analyzer integrated in Android Studio [6]. Lint integrates a set of rules that can generate patches of CC issues.

All the experiments were conducted on a server with 64-core Intel Xeon CPU E5-2683 v4 @ 2.10GHz processor and 256GB RAM. We configured the target API level l_t as API level 33, the latest Android framework version when we conducted experiments. We use Android emulators with 4GB RAM, 8GB ROM, and a screen resolution of 1440x3120 for experiments to avoid inconsistencies induced by different hardware profiles.

Table 3: Statistics of 77 CC issues in 13 Android apps.

App Name	Category	Install	Stars	#Issue
airmessage [3]	Communication	100K+	120	1
Buran [11]	-	-	71	1
cwa-app-android [12]	Health & Fitness	10M+	2.5K+	3
FairEmail [23]	Communication	500K+	1.8K+	32
fenix [24]	Communication	100M+	6.6K+	18
LibreTube [21]	-	-	5.3K+	1
lichobile [22]	Board Games	5M+	1.8K+	1
MaterialFiles [31]	Tools	100K+	3.4K+	1
Music-Player-Go [14]	Music & Audio	100K+	1.4K+	7
Neko [10]	-	-	1.6K+	1
organicmaps [25]	Travel	100K+	5.1K+	1
ProtonMail [26]	Communication	5M+	1.5K+	8
ProtonVPN [27]	Tools	10M+	1.4K+	2
Total				**77**

Table 4: Evaluation results of ConfFix and baselines

Tool	Pls	Crt	Sem	Syn	O	Suc	P
ConfFix	**320/64**	**318/64**	**290/58**	**60/12**	**2/1**	**64**	**100.0%**
Lint	**32/32**	**32/32**	**32/32**	**1/1**	**0/0**	**32**	**100.0%**
XFix	**200/40**	**15/3**	**15/3**	**0/0**	**185/37**	**3**	**7.5%**

X/Y: X is the number of patches in this category; Y is the number of issues to which the patches belong. **Pls** denotes the number of plausible patches. **Crt** denotes the number of correct patches. **Sem** denotes the number of issues that preserve the same semantics. **Syn** denotes the number of syntactically-equivalent patches. **O** denotes the number of overfitting patches. **Suc** denotes the number of issues successfully repaired (no overfitting patches achieve the highest fitness score in five runs). **P** denotes the precision.

5.2 RQ1: Effectiveness

As the results of ConfFix are subject to randomness, we ran ConfFix five times on the 77 real-world CC issues. We set the time budget of ConfFix to 120 minutes per run, following the existing practice [61]. We validated the patched app app_p as follows. Specifically, we first checked if app_p preserves the same runtime behavior at API level l_t with eliminating inconsistencies at API level l_i. We then explored the functionalities of the activities that render e_i and checked if app_p introduces any unintended runtime behavior (e.g., crashes). A patch is labeled as **correct (C)** if it satisfies the above conditions on the Android emulators between API level l_i and l_t; otherwise, the patch is labeled as **overfitting (O)**.

The results are shown in Table 4. ConfFix generated 320 plausible patches for 64 issues (Column **Pls**). Of the 320 plausible patches, 318 are the correct patches for 64 issues (Column **Crt**). ConfFix successfully repaired these 64 issues (i.e., no overfitting patches achieve the highest fitness score in five runs) with a precision of 100.0% (64/64). This suggests that ConfFix can save the app developers' debugging efforts of validating plausible patches.

We checked the semantics of correct patches (denoted as **Sem**) generated by ConfFix. Specifically, by referring to the API documentation in Android Developers [6], we checked if there is any behavioral difference between A_f and A_i (i.e., semantically equivalent). As shown in Table 4, among 318 correct patches, 290 of them for 58 issues leverage A_f with the same semantics as A_i. This is consistent with our finding that app developers prefer issue-fixing attributes that are semantically equivalent with issue-inducing attributes (See Section 3). Moreover, ConfFix output 60 patches that are syntactically equivalent to human patches in 12 issues (denoted as **Syn**). The above results suggest that ConfFix can generate patches that meet developers' expectations. Among the successfully repaired CC issues, ConfFix consistently outputs the same correct patches for 60 issues in five runs. ConfFix successfully generated multiple correct patches for four issues in our evaluation dataset. As in the example discussed in Section 2, ConfFix generated two correct patches by `app:lineHeight` and `android:lineSpacingExtra`. These two attributes share the same semantics as `android:lineHeight` by adjusting the spacing between two lines of text.

ConfFix generated two overfitting patches (Column **O**) for one issue. This is because of the randomness of ConfFix when searching for a numerical attribute value in order to achieve the optimal fitness score. For instance, cwa-app-android's commit 8296d1f [13] files an issue induced by `android:gravity` not available at an API level ≤ 23. While ConfFix can generate correct patches by specifying `android:top="36dp"`, ConfFix can also generate incorrect patches (e.g., `17dp`) due to the randomness of attribute value searching process. ConfFix successfully repaired this issue by prioritizing the correct patches over the overfitting patches.

ConfFix failed to generate plausible patches for 13 issues. Specifically, ConfFix failed to identify the key objects for two CC issues (See Section 4.1). For the remaining issues, ConfFix failed to recommend issue-fixing attributes to eliminate inconsistencies in key fields. For example, fenix's commit 7cea2ed [16] files an issue caused by inconsistent handling of `android:background` for API levels ≤ 23. ConfFix failed to generate a plausible patch because there are no other attributes in the Android framework that can affect the key fields of `android:background` (e.g., `mBackground`).

5.3 RQ2: Comparison

To answer RQ2, we ran Lint and XFix on 77 CC issues. We repeated XFix five times since the results of XFix are subject to randomness. We set the time budget of XFix to 120 minutes, the same as ConfFix.

The experimental results are shown in Table 4. For the 77 CC issues, Lint generated 32 patches that preserve the same semantics as the issue-inducing attributes (Column **Sem**). One correct patch generated by Lint is syntactically equivalent to human patches (Column **Syn**). Lint achieves a precision of 100.0% (32/32) by successfully repairing 32 issues, which can also be successfully repaired by ConfFix in consistently generating the same correct patches as Lint. However, Lint failed to generate plausible patches for the remaining issues because Lint does not contain rules to provide suggestions on fixing them (See Section 2). In particular, the 32 issues that Lint successfully repaired are all induced by `android:drawableTint`.

XFix generated 200 plausible patches for 40 issues, of which 15 patches for three issues are correct, while preserving semantics as the issue-inducing attributes. XFix successfully repaired three issues (Column **Suc**) with a precision of 7.5% (3/40). These three issues can also be successfully repaired by ConfFix by generating the same correct patches as XFix. XFix generated 185 overfitting patches for 37 issues, and failed to generate patches for the rest of issues. The reason is that XFix works by adjusting the UI layout bounds rendered by issue-inducing attributes, and these CC issues do not exhibit inconsistent UI layout bounds across API levels. For example, FairEmail [23] specified `android:drawableTint`, which was introduced at API level 23 and can induce inconsistent icons' color at lower API levels. XFix failed to handle such cases as the CC issue induced by `android:drawableTint` does not manifest the inconsistent UI layout bounds across API levels.

The above results show that ConfFix outperforms baselines in terms of the number of CC issues that were successfully repaired. As discussed in Section 2, for each CC issue, ConfFix works by diagnosing how the Android framework code changes can induce inconsistent UI displays across API levels, and then generating patches that can eliminate such inconsistencies. By doing so, ConfFix can generate correct patches without modeling every single visual feature that can be manifested by CC issues.

5.4 RQ3: Usefulness

ConfFix successfully repaired 40 CC issues that have not yet been fixed in the latest version of Android apps. We submitted issue reports or pull requests for these previously-unknown issues to seek app developers' feedback. In 36 out of 40 issues, ConfFix generated one correct patch, which was submitted to app developers for their feedback. For the remaining four issues, ConfFix generated a variety of correct patches with the highest fitness scores. In this case, we submitted patches whose A_f are semantically-equivalent (**Sem**) as A_i (See Section 5.2) For each of the issues, ConfFix produced only one patch that satisfies the above condition.

We **complied with the app projects' contributing guidelines and licenses** to submit pull requests and issue reports. We submitted issue reports only when the projects did not accept any external pull requests for copyright reasons. To help developers understand the reported CC issues and patches, we provided the issue reproduction steps and the screenshots concerning inconsistent UI behavior of CC issues. Additionally, we have thoroughly tested the patches to avoid spamming the open-source community.

In total, we submitted patches for 39 CC issues, 38 of which were confirmed and merged by app developers by the time of paper submission. We did not submit the patch for one issue, as the developer fixed it before we filed a pull request. The developers' patch for this issue is syntactically equivalent to the one generated by ConfFix. One remaining patch is still awaiting the app developers' decision. So far, we have not received a rejection from app developers. The high response rate (38/39=97.5%) is attributed to the attached screenshots and reproduction steps. As commented by the developers of FairEmail [23], the attached screenshots and issue reproduction steps are essential as the issue-inducing attributes were intended to fix other bugs. We also received positive comments from app developers. Below are two examples:

- *"Thank you, this works great!"*, in the pull request #45 of airmessage [2].
- *"Looks great! Thank for the investigation and fix!"*, in the pull request #26364 of fenix [17].

6 THREATS TO VALIDITY

Empirical Dataset. In this paper, we empirically studied the developers' practices of patching CC issues in Android apps. We further propose ConfFix to repair such CC issues automatically based on the empirical findings. Our findings are based on the state-of-the-art empirical dataset that was released by Huang et al. [43] in November 2021. Although the empirical dataset does not contain CC issues at the latest API level (API level 33 when the paper was submitted), experimental results show that the findings distilled from the dataset are also applicable for the latest API level.

Evaluation Subject Selection. We evaluated ConfFix on a set of 77 CC issues in 13 open-source Android apps. However, our results may not be generalizable beyond these apps. Nevertheless, these apps were selected from a large set of candidate apps in F-Droid [15] and GitHub [18] (See Section 5.1). We carefully inspected each commit returned by the keyword-based search and collected valid ones without bias. We evaluated ConfFix on open-source Android apps as we can access their issue trackers and source code. Developers can also deploy ConfFix to closed-source Android apps when the source code and UI test scripts are available.

Subjectivity of Manual Inspections. In this paper, we manually inspected the patches in the empirical dataset to obtain our findings. We also performed manual validation on patches generated by ConfFix and baselines. The manual checking can be subject to mistakes. To mitigate this threat, two authors participated in the manual process to ensure that the results were consistent. We released our dataset and evaluation results for public access.

7 RELATED WORK

Android Compatibility Issues. Compatibility issues are considered a major challenge for app developers [45]. Existing studies [35, 42, 43, 56, 58, 60] have explored the characteristics of compatibility issues from different perspectives. For example, Wei et al. [56, 58] studied the common practices of app developers to test, diagnose, and repair fragmentation-induced compatibility issues. Huang et al. [42] analyzed how callback APIs can induce compatibility issues in Android apps. Cai et al. [35] studied compatibility issues occurring at both the installation and the run time. Xia et al. [60] studied the practice of developers to handle Android compatibility issues from the large-scale app code base. Recently, Huang et el. [43] performed the first empirical study to understand how code changes in the Android framework can induce CC issues. However, these studies do not analyze developers' strategies in patching CC issues to facilitate automatic repair.

Detecting compatibility issues in Android apps has been widely studied in the research community [37, 41–43, 46, 48–50, 56–58]. First, DiffDroid [37] and Mimic [46] are dynamic-based approaches to test compatibility issues by comparing app UI differences among Android devices. Second, many existing static-based approaches [41–43, 48, 49, 56–58] use predefined rules to detect compatibility issues in Android apps. Among them, ConfDroid proposed by Huang et al. [43] is the state-of-the-art approach for detecting CC issues for Android apps. ConfDroid is driven by the rules extracted from the Android framework code changes. However, ConfDroid cannot provide suggestions on CC issue repair. To fill the research gap, we propose ConfFix, which leverages the knowledge learned from the Android framework to facilitate CC issue repair.

Existing studies [38–40, 47, 62] have been proposed to automatically repair compatibility issues in Android apps. For example, AppEvolve [38] and A3 [47] updates the usage of incompatible APIs based on the patterns learned from the examples of updating incompatible APIs in other apps. Recently, Zhao et al. proposed RepairDroid [62], which leverages the templates manually refined from the developers' examples to repair Android compatibility issues at the bytecode level. However, these approaches can only repair issues caused by problematic API invocations and are inapplicable for repairing CC issues in Android apps. Besides, Lint [20], a static analyzer officially released by Android, contains predefined rules for repairing CC issues. As revealed in Section 5, Lint failed to generate plausible patches for CC issues beyond the capabilities of those predefined rules. To fill the above research gaps, we empirically analyzed the common practices of app developers in patching CC issues. We then proposed ConfFix that can automatically repair CC issues based on the empirical findings.

Visual-based Issue Repair. There are existing approaches that focus on repairing issues related to visualization in GUI-based software [32, 33, 36, 51, 52, 54, 59]. For example, MFix [52] and MobileVisFixer [59] focuses on repairing mobile-friendly issues in web pages. CBRepair proposed by Alameer et al. [32] focuses on repairing internationalization presentation issues in web pages. LabelDroid [36], COALA [54] and SALEM [33] focus on improving the accessibility for mobile apps. XFix [51] focuses on repairing layout incompatibilities across web browsers based on a predefined set of visual features. However, XFix is ineffective in repairing CC issues whose inconsistent visual features go beyond the predefined rules. ConfFix fills the research gap by leveraging the knowledge learned from the Android framework to repair CC issue.

8 CONCLUSION & FUTURE WORK

In this paper, we empirically studied developers fixing practices on 196 CC issues in Android apps. We further propose ConfFix, which generates patches to eliminate any inconsistent handling of issue-inducing attributes across API levels. The evaluation results show the effectiveness of ConfFix in generating patches that are in line with app developers' expectations. Currently, ConfFix is driven by manually built UI test scripts. In the future, we plan to design an approach that can generate UI test scripts to reproduce the CC issues.

ACKNOWLEDGMENT

We thank the anonymous reviewers on the feedback for this research work. This research work is supported by National Natural Science Foundation of China (Grant No. 61932021, No. 62002125), Hong Kong Research Grant Council/General Research Fund (Grant No. 16207120), Hong Kong Research Grant Council/Research Impact Fund (Grant No. R5034- 18), the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001), and Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562). Huaxun Huang is supported by Hong Kong PhD Fellowship Scheme.

REFERENCES

[1] 2023. AdAway commit 576720b. https://github.com/AdAway/AdAway/commit/576720b

[2] 2023. airmessage pull request #45. https://github.com/airmessage/airmessage-android/pull/45

[3] 2023. airmessage/airmessage-android. https://github.com/airmessage/airmessage-android

[4] 2023. AmazeFileManager commit b3b8d60. https://github.com/TeamAmaze/AmazeFileManager/commit/b3b8d60

[5] 2023. Android API Differences Report. https://developer.android.com/sdk/api_diff/s-dp2/changes/changes-summary

[6] 2023. Android Developers. https://developer.android.com/

[7] 2023. apps-android-wikipedia commit 4738471. https://github.com/wikimedia/apps-android-wikipedia/commit/4738471

[8] 2023. apps-android-wikipedia commit 53010ec. https://github.com/wikimedia/apps-android-wikipedia/commit/53010ec

[9] 2023. Available Resources. https://developer.android.com/guide/topics/resources/available-resources?hl=en

[10] 2023. CarlosEsco/Neko. https://github.com/CarlosEsco/Neko

[11] 2023. Corewala/Buran. https://github.com/Corewala/Buran

[12] 2023. corona-warn-app/cwa-app-android. https://github.com/corona-warn-app/cwa-app-android

[13] 2023. cwa-app-android commit 8296d1f. https://github.com/corona-warn-app/cwa-app-android/commit/8296d1f

[14] 2023. enricocid/Music-Player-GO. https://github.com/enricocid/Music-Player-GO

[15] 2023. F-Droid - Free and Open Source Android App Repository. https://www.f-droid.org/en/

[16] 2023. fenix commit 7cea2ed. https://github.com/mozilla-mobile/fenix/commit/7cea2ed

[17] 2023. fenix pull request #26364. https://github.com/mozilla-mobile/fenix/pull/26364

[18] 2023. GitHub. https://github.com

[19] 2023. Google Play. https://play.google.com/store/apps

[20] 2023. Improve your code with lint checks. https://developer.android.com/studio/write/lint

[21] 2023. LibreTube. https://github.com/libre-tube/LibreTube

[22] 2023. lichess-org/lichobile. https://github.com/lichess-org/lichobile

[23] 2023. M66B/FairEmail. https://github.com/M66B/FairEmail

[24] 2023. mozilla-mobile/fenix. https://github.com/mozilla-mobile/fenix

[25] 2023. organicmaps/organicmaps. https://github.com/organicmaps/organicmaps

[26] 2023. ProtonMail/proton-mail-android. https://github.com/ProtonMail/proton-mail-android

[27] 2023. ProtonVPN/android-app. https://github.com/ProtonVPN/android-app

[28] 2023. Top Android SDK Versions. https://www.appbrain.com/stats/top-android-sdk-versions

[29] 2023. Update the order notes design. https://github.com/mozilla-mobile/fenix/pull/15886

[30] 2023. Write automated tests with UI Automator. https://developer.android.com/training/testing/other-components/ui-automator

[31] 2023. zhanghai/MaterialFiles. https://github.com/zhanghai/MaterialFiles

[32] Abdulmajeed Alameer, Paul T Chiou, and William GJ Halfond. 2019. Efficiently repairing internationalization presentation failures by solving layout constraints. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST 2019)*. IEEE, 172–182. https://doi.org/10.1109/ICST.2019.00026

[33] Ali S Alotaibi, Paul T Chiou, and William GJ Halfond. 2021. Automated Repair of Size-Based Inaccessibility Issues in Mobile Applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*. IEEE, 730–742. https://doi.org/10.1109/ASE51524.2021.9678625

[34] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269. https://doi.org/10.1145/2771783.2771803

[35] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A large-scale study of application incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. 216–227. https://doi.org/10.1145/3293882.3330564

[36] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE 2020)*. 322–334. https://doi.org/10.1145/3377811.3380327

[37] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. 308–318. https://doi.org/10.1109/ASE.2017.8115644

[38] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis (ISSTA 2019)*. 204–215. https://doi.org/10.1145/3293882.3330571

[39] Stefanus A Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automatic Android deprecated-API usage update by learning from single updated example. In *Proceedings of*

the 28th international conference on program comprehension (ICPC 2020). 401–405. https://doi.org/10.1145/3387904.3389285

[40] Stefanus A Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. 2022. AndroEvolve: Automated Android API update with data flow analysis and variable denormalization. *Empirical Software Engineering (EMSE 2022)* 27, 3 (2022). https://doi.org/10.1007/s10664-021-10096-0

[41] Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. 2018. Understanding and detecting evolution-induced compatibility issues in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. 167–177. https://doi.org/10.1145/3238147.3238185

[42] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. 532–542. https://doi.org/10.1145/3238147.3238181

[43] Huaxun Huang, Ming Wen, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing and Detecting Configuration Compatibility Issues in Android Apps. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*. IEEE, 517–528. https://doi.org/10.1109/ASE51524.2021.9678556

[44] Stéphane Jacquet, Xavier Chamberland-Thibeault, and Sylvain Hallé. 2021. Automated Repair of Layout Bugs in Web Pages with Linear Programming. In *International Conference on Web Engineering (ICWE 2021)*. 423–439. https://doi.org/10.1007/978-3-030-74296-6_32

[45] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real challenges in mobile app development. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 15–24. https://doi.org/10.1109/ESEM.2013.9

[46] Taeyeon Ki, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2019. Mimic: UI compatibility testing system for Android apps. In *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*. 246–256. https://doi.org/10.1109/ICSE.2019.00040

[47] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering* 48, 2 (2020), 417–431. https://doi.org/10.1109/TSE.2020.2988396

[48] Cong Li, Chang Xu, Lili Wei, Jue Wang, Jun Ma, and Jian Lu. 2018. ELEGANT: Towards Effective Location of Fragmentation-Induced Compatibility Issues for Android Apps. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC 2018)*. 278–287. https://doi.org/10.1109/APSEC.2018.00042

[49] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. 153–163. https://doi.org/10.1145/3213846.3213857

[50] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. 2022. Automatically Detecting API-induced Compatibility Issues in Android Apps: A Comparative Analysis (Replicability Study). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. 617–628. https://doi.org/10.1145/3533767.3534407

[51] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2017. Automated repair of layout cross browser issues using search-based techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. 249–260. https://doi.org/10.1145/3092703.3092726

[52] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2018. 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)*. 215–226. https://doi.org/10.1109/ICST.2018.00030

[53] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William GJ Halfond. 2021. Effective automated repair of internationalization presentation failures in web applications using style similarity clustering and search-based techniques. *Software Testing, Verification and Reliability* 31, 1-2 (2021). https://doi.org/10.1002/stvr.1746

[54] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. 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 (ESEC/FSE 2021)*. 107–118. https://doi.org/10.1145/3468264.3468604

[55] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*. 210–225. https://doi.org/10.1007/978-3-642-37057-1_15

[56] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 226–237. https://doi.org/10.1145/2970276.2970312

[57] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019)*. IEEE, 878–888. https://doi.org/10.1109/ICSE.2019.00094

[58] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2020. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering (TSE 2020)* 46, 11 (2020), 1176–1199. https://doi.org/10.1109/TSE.2018.2876439

[59] Aoyu Wu, Wai Tong, Tim Dwyer, Bongshin Lee, Petra Isenberg, and Huamin Qu. 2020. Mobilevisfixer: Tailoring web visualizations for mobile phones leveraging an explainable reinforcement learning framework. *IEEE Transactions on Visualization and Computer Graphics (TVCG 2020)* 27, 2 (2020), 464–474. https://doi.org/10.1109/TVCG.2020.3030423

[60] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, and Zhemin Yang. 2020. How Android Developers Handle Evolution-induced API Compatibility Issues: A Large-scale Study. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. 886–898. https://doi.org/10.1145/3377811.3380357

[61] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–670. https://doi.org/10.1109/ASE.2017.8115676

[62] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards Automatically Repairing Compatibility Issues in Published Android Apps. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*. 2142–2153. https://doi.org/10.1145/3510003.3510128

Received 2023-02-16; accepted 2023-05-03