ICSE: G: Woodpecker: Identifying and Fixing UI Display Issues in Mobile Applications

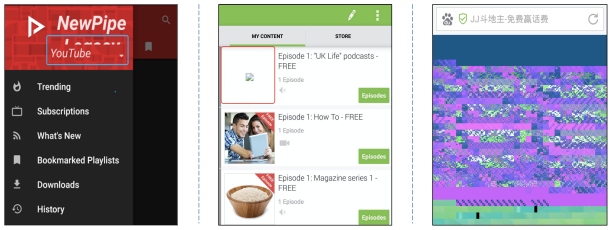
Zhe Liu
Institute of Software Chinese Academy of Sciences, University of Chinese Academy of Sciences

ABSTRACT

The complexity of GUI and some combinations of personalized settings make the UI display issues occur frequently. Unfortunately, little is known about the causes of UI display issues. The Android fragmentation and variety of UI components post a great challenge to repair the issue. Based on the our empirical study, this paper proposes `Woodpecker` to automatically detect, localize and repair UI display issues in Android apps. It detects screenshots with issues with computer vision technology, localizes buggy source code from screenshot, and repairs issues with pre-defined templates automatically. We evaluate `Woodpecker` with 30 real-world UI display issues, it can successfully detect 87% and repair 91% issues. We further apply `Woodpecker` to another 316 popular open-source Android apps, and successfully uncover 123 previously-undetected issues. It can automatically repair 116 (94%) issues, with 106 of them accepted by developers so far, while others pending (none of them is rejected). The `Woodpecker` demo video link: https://youtu.be/uPXEtqaIP8g.

1 RESEARCH PROBLEM AND MOTIVATION

With the evolution of Android apps [1, 2], graphical user interface (GUI) is one of the most important ingredients in app development. A good UI design can make the app easy, practical and efficient to use, which significantly affects the success of the app and the loyalty of its users [17, 19]. However, more and more fancy visual effects in UI design such as intensive media embedding, animation, light, floating and shadows post a great challenge for developers in the implementation. Due to the GUI complexity, many display issues including *occlusion and overlap, missing and incorrect information* and *blurred screen* [10, 27] (as shown in Fig 1) occur frequently which negatively influence the fluent usage with the app, resulting in the significantly bad user experience and corresponding loss of users. While detecting display issues [10, 27, 53] is a necessary step toward improving GUI, it leaves a harder task of issue repair unsolved [18, 20, 38, 39].

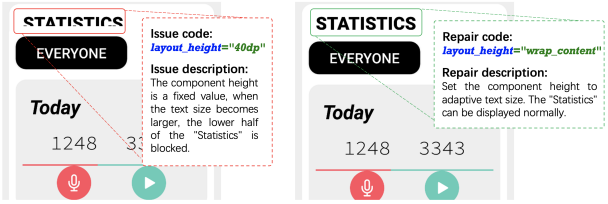


(a) Occlusion and overlap (b) Missing and incorrect information (c) Blurred screen

Figure 1: Examples of three types of UI display issues

The difficulty of UI display issue repair is compounded by the software complexity with legacy code, code inconsistency, and code amount, especially in highly fragmented front-end code [5, 7, 18]. The developer must understand the problem, replay the issue, localize its root cause in the source code, and speculate about strategies to possibly repair it. Unlike the web UI, the rendering mechanism of Android UI is more complex, which requires more accurate issue localization and patching. Although there are many automated issue repair methods in Android crash bug and Web UI [8, 12, 31–35] based on genetic algorithm [18, 44, 46], search method [28, 41, 47], repair patterns [26, 55], and semantic information [14, 36, 50], none of them is patching Android UI display issues.

To tackle these challenges, we propose an approach to automate the whole process for reducing developers' burden. Before doing that, we conduct the first empirical study in Android UI display issues about the root causes and repair strategies from 4,230 bug reports and construct the dataset. Through the manual categorization, we summarize 10 categories of root causes, and corresponding repair strategies, in terms of the three types of the UI display issues. In Fig 2 the occlusion and overlap issues can be caused by the unadaptable boundary or layout settings, and the common repair strategy is to employ an adaptive boundary. These findings facilitate developers in understanding the issues, and serve as the basis for the design of following repair method.



(a) Buggy UI screenshot (b) Fixed UI screenshot

Figure 2: Example of UI display issue repair

Based on the observations, we propose `Woodpecker`[1] to automatically detect, localize, and repair the UI display issues in Android apps. It mainly includes three parts: detecting UI display issues in screenshots, localizing issues in source code, and repairing issues with templates. `Woodpecker` takes the source code and APK of an app as input, automatically runs the app, and obtains the screenshots and corresponding view hierarchy files. It employs computer vision technology to detect issues in screenshots and highlights the buggy area. Then it obtains the coordinates of the buggy area, and localizes the buggy source code with the help of the detailed component information in the view hierarchy file. Finally, it repairs the issues by applying 9 pre-defined templates automatically.

We evaluate `Woodpecker` on 30 real-world UI display issues, and achieve the accuracy score of issue detection as 87% and repair as 91%. We further apply it to another 316 popular Android apps, and it successfully detects 123 previously unknown UI display issues. It can automatically repair 116 (94%) issues, and we submit these patches in Github pull requests, among which 106 are merged by developers so far, the rest are pending, and none of them is rejected.

[1] Our approach is named as `Woodpecker` as it is like the woodpecker (forest doctor) to effectively detect, localize and repair UI display issues.

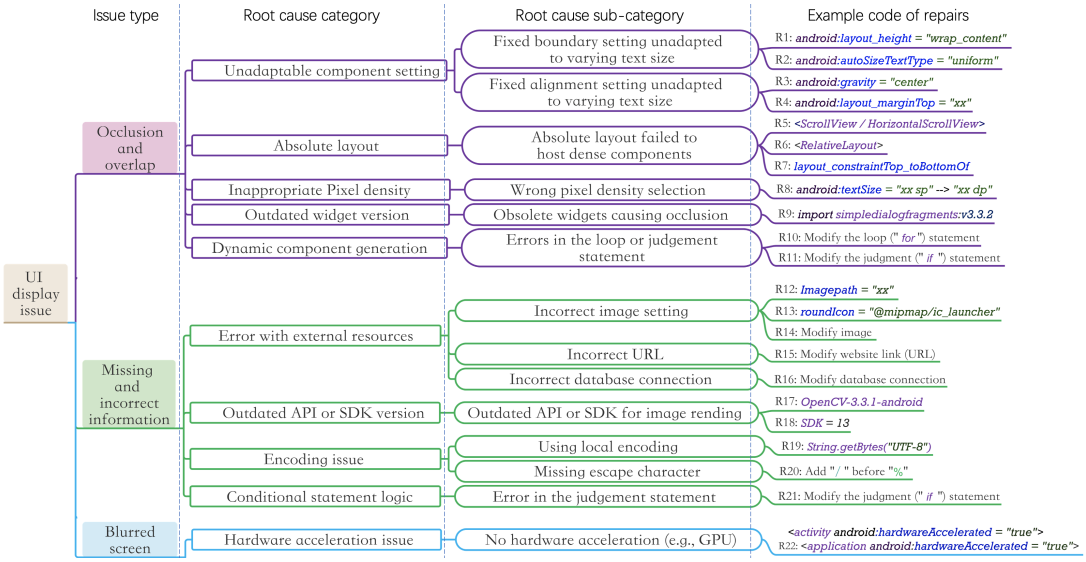


Figure 3: Root causes and repair strategies of UI display issues and corresponding occurrence frequency

2 BACKGROUND AND RELATED WORK

To ensure that mobile apps are working well, many static and automatic GUI testing tools to detect bugs, stylistic errors, and suspicious constructs [3, 4, 9, 40, 42, 45, 48, 49, 51, 54, 56]. Most of these existing studies targeted at the functionality issues as app crash. Recently, researchers shifted their focus on UI related issues to improve the usability and accessibility of mobile apps, such as UI rendering delays [16], image loading [24] and UI display issues [27]. This paper also focuses on the UI display issues to further disclose its root causes and repair approach.

Due to the uncertainty of test case generation [44], existing automatic program repair methods [11, 14, 21, 30, 36, 50, 52] can not be directly used for issue repair of mobile apps. For this reason, several automatic repair researches [13, 15, 20, 22, 26, 29, 44] for mobile apps used template-based issue repair method to fix crash bugs. Different from them, we target specifically at UI display issues with an automated approach in both repair and detection.

Except for Android issue, research in Web UI [8, 12, 31–35] compared the differences between current UI and oracle UI for bug detection, and use the search-based method to repair them. However, Android UI has a more complex rendering mechanism, which can't modify CSS to repair bugs and check the correctness in real time like Web UI. Android can only modify the source code and compile and render it before checking its correctness. Therefore, repairing Android UI requires more accurate detection and location.

3 EMPIRICAL STUDY OF UI DISPLAY ISSUES

3.1 Dataset Collection and Categorization

To examine the mobile UI display issues, we search two popular open-source Android app hosting websites: F-Droid and GitHub to investigate the root causes and repair strategies. We employ the following criteria for app selection: more than 1K downloads in Google Play (popular), public issue tracking system (traceable) and more than 100 code commits (well maintained). And a total of 2,566 open-source Android apps are selected in this study.

Two researchers then manually check the above-returned candidate issue reports to ensure that the reports are truly related to the UI display issues. The third researcher reexamines the results until a consensus is reached. After such inspection, we collect 4,230 issue reports with UI display issues. These issue reports constitute the dataset of our empirical study. With the retrieved UI display issue reports, we conduct manual categorization to derive the root causes and repair strategies of the UI display issues. Following the open coding protocol [43], they analyze the issue reports and categorize root causes of UI display issues and repair strategies.

3.2 Why Cannot Mobile UI Display Correctly?

3.2.1 ***Causes of Occlusion and Overlap Issues.*** There are 72% of the investigated issues belonging to this type. These issues can be manifested by setting the *Font size* or *Display size* as *largest*. They mainly appear in the *TextView*, *EditText* and *Button* components. We summarize five categories (**Unadaptable component setting**, **Absolute layout**, **Inappropriate Pixel density**, **Outdated widget version**, **Dynamic component generation**) of root causes as shown in Figure 3.

3.2.2 ***Causes of Missing and Incorrect Information Issues.*** We observe 26% of the examined reports related to the missing and incorrect information issues. These issues mainly appear in the *ImageView* and *TextView* components. We summarize four categories (**Error with external resources**, **Outdated API/SDK version**, **Encoding issue**, **Conditional statement issue**) of root causes for these issues, as shown in Figure 3.

3.2.3 ***Causes of Blurred Screen Issues.*** 2% of the issue reports are about blurred screens. The main reason is Android hardware acceleration is not turned on. Many apps can obtain good fluency of graphics rendering by using the device GPU, which requires the hardware acceleration as open. When it is turned off, the screen can easily get stuck, resulting in a blurred screen.

3.3 How Do Developers Fix UI Display Issues?

We found that 82% of the issues need to repair the XML file, and the rest need to repair the Java file. In the following paragraphs, we use the *Rx* to refer to the example issue repair code in Figure 3.

3.3.1 ***Repair Strategies of Occlusion and Overlap Issues.*** The common repair strategies for the issues caused by unadaptable component settings are to employ adaptive boundary (R1, R2) or centered alignment (R3). For the absolute layout, the general repair strategies are to modify the layout to *ScrollView* or *RelativeLayout* (R5, R6). Besides, some developers would directly modify the parameters in the original alignment or layout setting (R4, R7), which is less flexible. The developers would modify *sp* to *dp* (R8) to tackle the issues caused by inappropriate pixel density. Utilizing a newer version (R9) can repair the issues caused by the outdated widget version, for issues caused by dynamic component generation, one would modify the defective code (R10, R11) to let it work properly.

3.3.2 ***Repair Strategies of Missing and Incorrect Information Issues.*** The common repair strategy for issues caused by the error with external resources is to utilize the correct external resources as shown in R12 to R16. Upgrading the API/SDK version (R17, R18) can repair the issues caused by outdated API/SDK version. For the encoding issue, developers employ the universal encoding or add the escape character to let the information display correctly (R19, R20). In addition, developers would modify the defective code to repair the issues caused by conditional statement logic.

3.3.3 ***Repair Strategies of Blurred Screen Issues.*** The repair strategy for this type is to turn on the hardware acceleration. Both the application level and activity level hardware acceleration are enabled to facilitate the graphics rendering.

4 APPROACH AND UNIQUENESS

Based on the observations from the empirical study, we propose `Woodpecker` to automatically detect, localize, and repair the UI display issues in Android apps in Fig 4. Note that, the reason we should conduct the issue detection and localization before auto-repair is because the UI display issues are caused by complex control or data flow in the source code, and directly applying the repair templates can generate numerous false positives.

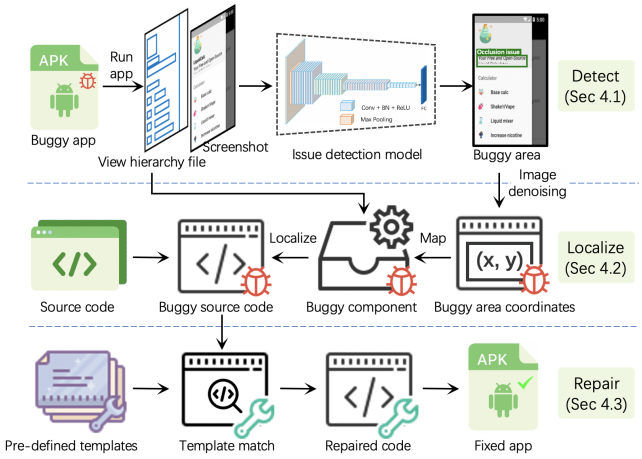


Figure 4: Overview of `Woodpecker`

4.1 Detect UI Display Issues in Screenshots

Because the display issues in UI can be easily detected by visual information, we adopt the object detection model for issue detection, which includes a feature extraction network (50 convolutional layers), a regional proposal network (RPN) module, and an ROI pooling module. It uses our empirical study data for model training. We also propose a GAN-based data generation approach to improve the performance. Specially, according to the feature map obtained by feature extraction module and proposal obtained by RPN module. It will input them into the ROI pooling layer to visualize the regions. Finally, `Woodpecker` will output the bounding box of buggy area.

4.2 Localize Issues in Source Code

After issue detection, we localize the visual issues in source code which lays a foundation for auto-repair. Since we can only retrieve the position of the issues in the screenshot which could not be directly mapped to the source code, we employ the view hierarchy file to act as a bridge between them, and design a two-phase issues location method.

We first fetch the coordinates of the buggy bounding box obtained from the screenshot. Meanwhile, we obtain the coordinates of all components in the screenshot from the view hierarchy file, which serve as the candidate localization targets. The obtained buggy bounding box might not perfectly match the actual buggy component. Hence we apply a relatively loose matching criterion, i.e., if the intersection area of the buggy bounding box and the candidate component is larger than 65% (through a pilot study) of the buggy bounding box, we treat the specific candidate component as the real buggy component. We then obtain the ID, type and text description of the buggy component from the view hierarchy file which serves as input for the second phase localization.

To further locate the buggy code snippet in source code, we use information retrieval based bug localization method [23, 37]. The utilized information includes buggy component ID, component type, and text description. In detail, we search source code for retrieving code snippets which match the *component ID*. For code snippets which successfully match, we then check whether the *component type* can also match the retrieved code snippets successfully. For some components without *component ID*, we use the *text description* and *component type* of component for retrieval. If more than one code snippet matches successfully, all of them are returned.

4.3 Repair Issues with Templates

In Fig 4, the issue repair would take the localized buggy source code as input, and use pre-defined templates to match the buggy code and repair the issue. Based on our empirical study about how developers repair UI display issues, `Woodpecker` defines 9 pre-defined templates (With the 9 templates, 86% UI display issues can be covered based on the statistics in empirical study). It checks whether the buggy code matches any of these 9 templates. Then `Woodpecker` will generate a corresponding patch from the template and automatically package the app to repair the UI display issue.

Template 1: Apply adaptive boundary setting. As shown in the code below, change the attribute of *layout_height* and *layout_width* to *wrap_content*, by which the component can resize automatically to accommodate a larger text font.

```
-   android:layout_width = "([\w\W]*)"
-   android:layout_height = "([\w\W]*)"
+   android:layout_width = "wrap_content"
+   android:layout_height = "wrap_content"
```

Template 2: Apply self-adjusting text setting. Change the attribute of *autoSizeTextType* to "*uniform*" so that the text can adjust its size with the changing component.

```
-   android:autoSizeTextType = "none"
+   android:autoSizeTextType = "uniform"
```

Template 3: Apply center alignment. Change the attribute of *gravity* to "*center*" to always center the contents of a component. Thereinto, **([\w\W]*)** is a regular expression to match such attributes as "top", "bottom", "left", "right".

```
-   android:gravity = "([\w\W]*)"
+   android:gravity = "center"
```

Template 4: Apply scroll view. Insert the *ScrollView* around the error *Linearlayout* snippet with which the component can auto-scroll the contents if necessary. Thereinto, *ScrollView* adopts its default setting.

```
-   <Linearlayout ...>
-       ....
-   </Linearlayout>
+   <ScrollView ...>
+       <Linearlayout  ...>
+           ....
+       </Linearlayout>
+   </ScrollView>
```

Template 5: Change pixel density. Change the attribute of *textSize* from "(^\d+$)sp" to "(^\d+$)dp", by which text size of component would not change with user-defined settings. Thereinto, (^\d+$) is a regular expression representing an arbitrary value.

```
-   android:textSize = "(^\d+$)sp"
+   android:textSize = "(^\d+$)dp"
```

Template 6: Update outdated widgets. We summarize 5 widgets that might trigger issues from the empirical study in Section 3.2, e.g., AppIntro, SimpleDialogFragments. This template will change the bug-prone version of the widget to a bug-free version. The full list of this template is on our website.

```
-   github.eltos:simpledialogfragments:v"[0-3](\.\d+)?"
+   github.eltos:simpledialogfragments:v3.3.2
```

Template 7: Update outdated API. We summarize 2 APIs which might trigger UI display issues from the empirical study in Section 3.2, i.e., OpenCV, Marked. This template will change the bug-prone version of the API to a bug-free version. The full list of this template is on our website.

```
-   opencv 3.3.0
+   opencv 3.3.+
```

Template 8: Apply universal encoding. Change the encoding to *UTF-8*, by which the text can be displayed correctly no matter the language setting of the mobile device.

```
-   String.getBytes("GBK");
+   String.getBytes("UTF-8");
```

Template 9: Turn on hardware acceleration. Change the app and activity level of *hardwareAccelerated* to *true* to avoid the blurred screen.

```
-   <application android:hardwareAccelerated="false">
+   <application android:hardwareAccelerated="true">
```

5 EVALUATION AND CONCLUSION

We evaluate the effectiveness of `Woodpecker` via experiments on its three main parts i.e., issue detection, localization, and repair. The evaluation dataset contains 30 issues from 30 different open-source Android apps in GitHub. `Woodpecker` achieves the accuracy score of issue detection as 87%, localization as 85% and repair as 91%. We further analyze the issues that cannot be repaired. The main reason is that the `Woodpecker` fails to localize the buggy code [25].

We further assess the usefulness of `Woodpecker` in real-world practice and randomly sample another 316 Android apps from F-Droid [6]. Given collected GUI screenshots, `Woodpecker` detects the UI display issues, and automatically generate repair patches for each detected issue. We then submit an issue report and a pull request with the generated repair patch for each issue. `Woodpecker` detects 123 previously undetected UI display issues from 107 apps, which again indicates the prevalence of UI display issues, and the necessity for auto-repair. It can repair 116 (94%) of them. For the 116 submitted pull requests, 106 of them are successfully merged by developers, while the remaining are still pending and none of them is rejected. 7 issues can't be repaired due to the error in logic code and external resources which are beyond the scope of our study. Note that, due to space limit, Table 1 presents only part of the merged pull requests, with the full list on our website. This high ratio of accepted pull requests further indicates the effectiveness and usefulness of `Woodpecker`.

Table 1: Samples of GitHub merged pull requests

No.	Commit ID	APP Name	Category	Version	Download
1	8a76280	AlarmClock	System	3.09.01	1M+
2	37c437c	GPSLogger	Navig	112	1M+
3	5482728	Syncthing	Internet	1.18.0	1M+
4	66c9c05	J2MELoader	Game	1.7.0	1M+
5	195b5ed	Openfood	Health	3.6.8	1M+
6	e5b359d	Goodtime	Time	2.2.7	500K+
7	703a262	PDFConverter	Media	8.8.1	100K+
8	8767e55	AppInt	Plugin	1.6.0	100K+
9	85b1ea7	Commons	Internet	3.0.3	50K+
10	5a3fb30	Openboard	System	1.4.3	50K+
11	eda6be6	EVMap	Navig	0.8.3	10K+
12	867b6e2	Hendroid	Reading	1.15.0	10K+
13	1eceeec	Minesweep	Games	12.4.2	10K+
14	d23ae3b	Democracy	Media	4.2.0	10K+
15	757a286	Easer	System	0.8.2	10K+
16	9567661	Watomatic	Internet	1.2.0	10K+
17	5225479	ChessClock	Games	2.5.0	10K+
18	e7a3b34	Look4Sat	Science	2.5.3	10K+
19	076f8e8	Frost	Media	2.1.2	10K+
20	eb6cfb1	Acrtions	Media	2.0.0	10K+

When merging the pull requests, some Android app developers express thanks such as "*Thank you very much for this improvement, this is great. Much appreciated!*" (i.e., 4345457, PicardScanner), "*Very nice! Thank you very much for this contribution!*" (i.e., 8a76280, AlarmClock). Furthermore, some developers also express their thought about the UI display issues "*I have to admit I didn't think of this kind of issue at first and after a bit of testing the problem seems to be a bit more complex beside these login screens.*" (i.e., 5fcf8e1, OpenWebWeaver).

In the future, we will keep looking for and fixing more types of GUI display issues with higher accuracy. And we will try to develop a more lightweight but efficient approach to locate the bug by scanning the source code with static analysis.

REFERENCES

[1] 2016. Mobile Internet use passes desktop for the first time. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/.

[2] 2020. Number of available applications in the Google Play Store from December 2009 to June 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store.

[3] 2021. http://tools.android.com/tips/lint.

[4] 2021. https://github.com/stylelint/stylelint.

[5] 2021. Common UI bugs. https://www.qamadness.com/common-user-interface-bugs-or-why-ui-testing-matters/.

[6] 2021. F-droid. http://f-droid.org/.

[7] 2021. GPSlogger. https://github.com/mendhak/gpslogger.

[8] Abdulmajeed Alameer, Paul T. Chiou, and William G. J. Halfond. 2019. Efficiently Repairing Internationalization Presentation Failures by Solving Layout Constraints. In *ICST 2019*. IEEE, 172–182. https://doi.org/10.1109/ICST.2019.00026

[9] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. *ICSE* (2020).

[10] Ke Chen, Yufei Li, Yingfeng Chen, Changjie Fan, Zhipeng Hu, and Wei Yang. 2021. Glib: towards automated test oracle for graphically-rich applications. In *FSE*. 1093–1104.

[11] Liushan Chen, Yu Pei, and Carlo Alberto Furia. 2020. Contract-based program repair without the contracts: An extended study. *TSE* (2020).

[12] Paul T. Chiou, Ali S. Alotaibi, and William G. J. Halfond. 2021. Detecting and localizing keyboard accessibility failures in web applications. In *ESEC/FSE '21*. ACM, 855–867. https://doi.org/10.1145/3468264.3468581

[13] Barthelemy Dagenais and Martin P Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. In *ICSE*. IEEE, 599–602.

[14] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*. 30–39.

[15] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing q&a sites (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[16] Yi Gao, Yang Luo, Daqing Chen, Haocheng Huang, Wei Dong, Mingyuan Xia, Xue Liu, and Jiajun Bu. 2017. Every pixel counts: Fine-grained UI rendering analysis for mobile applications. In *INFOCOM*. IEEE, 1–9.

[17] Linda Di Geronimo, Larissa Braz, Enrico Fregnan, Fabio Palomba, and Alberto Bacchelli. [n.d.]. UI Dark Patterns and Where to Find Them: A Study on Mobile Applications and User Perception.

[18] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[19] Bernard J. Jansen. 1998. The Graphical User Interface. *SIGCHI Bull.* 30, 2 (April 1998), 22–26. https://doi.org/10.1145/279044.279051

[20] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. 2019. Mining android crash fixes in the absence of issue-and change-tracking systems. In *ISSTA*. 78–89.

[21] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024.

[22] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *ESEC/SIGSOFT FSE 2019*. ACM, 314–325.

[23] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *2017 ICPC*. IEEE, 218–229.

[24] Wenjie Li, Yanyan Jiang, Chang Xu, Yepang Liu, Xiaoxing Ma, and Jian Lu. 2019. Characterizing and detecting inefficient image displaying issues in Android apps. In *SANER*. IEEE, 355–365.

[25] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 ICST*.

[26] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[27] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *2020 35rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[28] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.

[29] Siqi Ma, David Lo, Teng Li, and Robert H Deng. 2016. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 711–722.

[30] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *SANER*. IEEE, 468–478.

[31] Sonal Mahajan, Negarsadat Abolhassani, Phil McMinn, and William G. J. Halfond. 2018. Automated repair of mobile friendly problems in web pages. In *ICSE 2018*. ACM, 140–150. https://doi.org/10.1145/3180155.3180262

[32] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated repair of layout cross browser issues using search-based techniques. In *ISSTA*. ACM, 249–260. https://doi.org/10.1145/3092703.3092726

[33] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2018. Automated Repair of Internationalization Presentation Failures in Web Pages Using Style Similarity Clustering and Search-Based Techniques. In *ICST*. IEEE Computer Society, 215–226.

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

[35] Sonal Mahajan and William G. J. Halfond. 2015. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *ICST 2015*. IEEE, 1–10. https://doi.org/10.1109/ICST.2015.7102586

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[37] Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota, and Sonia Haiduc. 2018. Are bug reports enough for text retrieval-based bug localization?. In *2018 ICSME*. IEEE, 381–392.

[38] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[39] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *arXiv preprint arXiv:1802.02312* (2018).

[40] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2020. Testing of mobile applications in the wild: A large-scale empirical study on android apps. In *Proceedings of the 28th International Conference on Program Comprehension*. 296–307.

[41] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. 254–265.

[42] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. PATDroid: permission-aware GUI testing of Android. In *ESEC/FSE*. ACM, 220–232.

[43] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572.

[44] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *ICSE*. IEEE, 187–198.

[45] Mian Wan, Yuchen Jin, Ding Li, Jiaping Gui, Sonal Mahajan, and William G. J. Halfond. 2017. Detecting display energy hotspots in Android apps. *Softw. Test. Verification Reliab.* 27, 6 (2017). https://doi.org/10.1002/stvr.1635

[46] Z. Wang, J. Gao, X. Chen, F. U. Hao-Jie, and X. Y. Fan. 2018. Automatic Program Repair Techniques:A Survey. *Chinese Journal of Computers* 41, 3 (2018), 588–610.

[47] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*. IEEE.

[48] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *ISSTA*. ACM.

[49] Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2019. Analyses for specific defects in android applications: A survey. *Frontiers of Computer Science* (2019).

[50] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[51] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[52] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empir. Softw. Eng.* 23, 5 (2018), 2948–2979.

[53] Shengcheng Yu, Chunrong Fang, Yexiao Yun, Zhenfei Cao, Kai Mei, Zhihao Cao, and Zhenyu Chen. 2021. Detecting Crowdsourced Test Report Consistency for Mobile Apps with Deep Image Understanding and Text Analysis. *ICSE* (2021).

[54] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. 2022. Using code reviews to automatically configure static analysis tools. *Empir. Softw. Eng.* 27, 1 (2022), 28.

[55] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. 2021. Domain-Specific Fixes for Flaky Tests with Wrong Assumptions on Underdetermined Specifications. In *ICSE*. IEEE, 50–61.

[56] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-Based Linting of GUI Animation Effects Against Design-Don't Guidelines. In *ICSE*.