A User Interaction Bug Analyzer Based on Image Processing

Abel Méndez-Porras*
University of Costa Rica, Computer and Information Science Graduate Program
San José, Costa Rica
amendez@itcr.ac.cr

and

Jorge Alfaro-Velásco
Costa Rica Institute of Technology, Department of Computer Science
Ciudad Quesada, Costa Rica
joalfaro@itcr.ac.cr

and

Marcelo Jenkins
University of Costa Rica, Computer and Information Science Department
San José, Costa Rica
marcelo.jenkins@ecci.ucr.ac.cr

and

Alexandra Martínez Porras
University of Costa Rica, Computer and Information Science Department
San José, Costa Rica
alexandra.martinez@ecci.ucr.ac.cr

Abstract

Context: Mobile applications support a set of user-interaction features that are independent of the application logic. Rotating the device, scrolling, or zooming are examples of such features. Some bugs in mobile applications can be attributed to user-interaction features. **Objective:** This paper proposes and evaluates a bug analyzer based on user-interaction features that uses digital image processing to find bugs. **Method:** Our bug analyzer detects bugs by comparing the similarity between images taken before and after a user-interaction. SURF, an interest point detector and descriptor, is used to compare the images. To evaluate the bug analyzer, we conducted a case study with 15 randomly selected mobile applications. First, we identified user-interaction bugs by manually testing the applications. Images were captured before and after applying each user-interaction feature. Then, image pairs were processed with SURF to obtain interest points, from which a similarity percentage was computed, to finally decide whether there was a bug. **Results:** We performed a total of 49 user-interaction feature tests. When manually testing the applications, 17 bugs were found, whereas when using image processing, 15 bugs were detected. **Conclusions:** 8 out of 15 mobile applications tested had bugs associated to user-interaction features. Our bug analyzer based on image processing was able to detect 88% (15 out of 17) of the user-interaction bugs found with manual testing.

*Costa Rica Institute of Technology, Department of Computer Science, Costa Rica.

Keywords: bug analyzer, user-interaction features, image processing, interest points, testing

1 Introduction

The variety of mobile devices and their operating systems, known as *fragmentation* [1, 2, 3, 4], represents a testing challenge nowadays since mobile applications may behave differently regarding usability and performance depending on the device they are run on. Therefore, the number of tests required to verify that an application works as expected on existing mobile devices is increasingly high.

The behavior of smart mobile devices is highly interactive [5] and mobile applications are therefore affected by this interaction. Users perform actions on their applications and the applications respond to these actions. Several studies state the importance of the context where mobile applications are executed to ensure their quality [6, 7, 8]. According to Amalfitano and Fasolino [9], the quality of mobile applications is lower than expected due to rapid development processes where the activity of software testing is neglected or carried out superficially. To meet the growing demand for high quality applications, software testing and automation are key players. Zaeem et al. [10] conducted a study of defects in mobile applications, where they found that a significant fraction of bugs can be attributed to a class of features called *user-interaction*. They define a user-interaction feature as:

> An action supported by the mobile platform, which enables a human user to interact with a mobile app, using the mobile device and the graphical user-interface (GUI) of the app. Further, an interaction feature is associated with a common sense expectation of how the mobile app should respond to that action.

Common user-interaction features are: double rotation, killing and restarting, pausing and resuming, back button functionality, opening and closing menus, zooming in, zooming out, and scrolling. Mobile applications support user-interaction features associated with content presentation or navigation that are independent of the application logic. Automated testing of user-interaction features facilitates the search for defects in a variety of mobile applications.

In this paper, we propose a bug analyzer based on user-interaction features that detects bugs in mobile applications. Our approach raises a novel way of testing mobile applications to expose bugs caused by user-interaction features.

We seek to answer the following research questions:

RQ1: Based on user-interaction features, how often do bugs appear?

RQ2: How well can digital image processing detect bugs based on user-interaction features?

The main contributions of this paper are (*i*) the study of real mobile application bugs based on user-interaction features and (*ii*) the use of image processing to automatically detect user-interaction bugs.

This paper is organized as follows. Section 2 presents the background and related work, section 3 describes the bug analyzer in the context of an automated testing framework, section 4 explains the methodology, section 5 shows the findings and discussion, and section 6 presents the conclusions and future work.

2 Background and Related Work

In this section, we present an overview of Android activities, user-interaction features, and related work.

2.1 Android Activities

An *Activity* is an application component which gives the user a screen. In this screen, the user can interact with the application. An application usually consists of multiple activities that are loosely bound to each other. Typically, there is a main activity which is presented to the user when launching the application for the first time. One activity can start other activities. When a new activity starts, it takes the focus, and the previous activity is stopped and pushed onto the back stack [11]. Figure 1 shows the paths an activity might take between states. The rectangles represent the callback methods developers can implement to perform operations when an activity transitions between states. A brief description of each state is given next.

- onCreate: the activity is being created.
- onStart: the activity is being started.
- onResume: the activity is about to become visible.

Listing 1: Overriding the onPause method.

```
    @Override
    protected void onPause() {
        if(mCamera!=null){
            mCamera.release();
            mCamera=null;
        }

        userName.setText(``username");
}
```

- onPause: another activity is taking focus and the current activity is about to be "paused".
- onStop: the activity is no longer visible.
- onDestroy: the activity is about to be destroyed.

In association with other activities, the task activity and back stack affect the lifecycle of an activity. Implementing callback methods is very important to ensure the quality of an application. Listing 1 shows an example of the onPause method overriding. In this example, if the application was using the camera, it releases it since it will not be needed when paused and other activities may use it. The instruction where the userName object is set with a text string is an incorrect use of the onPause method, since the data contained in the userName object is lost when the activity is paused.

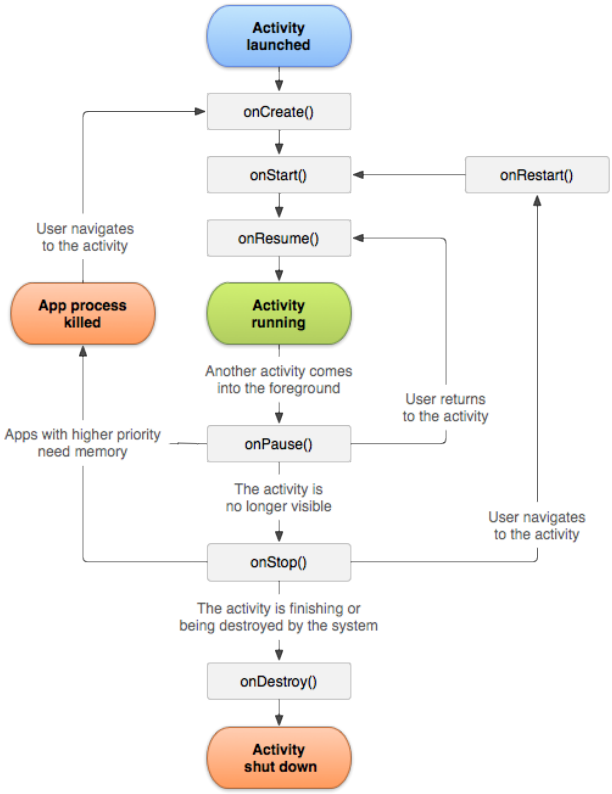

Figure 1: The activity lifecycle in Android from [11].

2.2 User-Interaction Features

Zaeem et al. [10] defined the following eight user-interaction features (abbreviations are shown in parenthesis):

1. Double rotation (dr): Rotating a mobile device and then rotating it back to the original orientation. With this action the application should stay in the same state.
2. Killing and restarting (kr): The operating system might choose to kill and then restart an app for various reasons (e.g., low memory). Similar to double rotation, the app should retrieve its original state and view.
3. Pausing and resuming (pr): The app can be paused (e.g., by hitting the Android Home button) and then resumed.
4. Back button functionality (bb): The Back button is a hardware button on Android devices which takes the app to the previous screen.
5. Opening and closing menus (menu): The hardware Menu button on Android devices opens and closes custom menus that each app defines.
6. Zooming in (zi): Zooming into a screen should bring up a subset of what was originally on the screen.
7. Zooming out (zo): Zooming out from a screen should result in a superset of the original screen.
8. Scrolling (scr): Scrolling down (or up) should display a screen that shares parts of the previous screen.

The user-interaction features mentioned above affect the states of the activities. Changes in the activities' states can generate bugs in the application behavior.

2.3 Related Work

The creation of automated testing tools has been increasing; here, some tools are described.

Dynodroid [12] uses model learning and random testing techniques for generating inputs to mobile applications. It is based on an "observe-select-execute" principle that efficiently generates a sequence of relevant events. It generates both user interface and system events.

MobiGUITAR [13] uses model learning and model based testing techniques. The application model is created using an automated reverse engineering technique called GUI Ripping. Test cases are generated using the model and test adequacy criteria. These test cases are sequence of events.

A^3E [14] uses model based testing techniques. This tool systematically explores applications while they are running on actual phones. A^3E uses a static, taint-style, data-flow analysis on the application bytecode in a novel way to construct a high-level control flow graph that captures legal transitions among activities (application screens). It then uses this graph to develop an exploration strategy named Targeted Exploration, and uses a strategy named Depth-first Exploration.

SwiftHand [15] generates sequence of test inputs for Android applications. It uses machine learning to learn a model of the application during testing, the learned model to generate user inputs that visit unexplored states of the application, and the execution of the applications on the generated inputs to refine the model.

Orbit [16] is based on a grey-box approach for automatically extracting a model of the application being tested. A static analysis of the application source code is used to extract the set of user actions supported by each widget in the GUI. Then, a dynamic crawler is used to reverse engineer a model of the application, by systematically exercising extracted actions on the live application.

3 The Bug Analyzer in Context

The bug analyzer presented here is a part of broader testing framework that we proposed in [17]. Such automated testing framework is also based on user-interaction features but includes more components than the bug analyzer. Figure 2 shows an overview of this framework, which is composed of an exploration environment, an inference engine, a bug analyzer, and a bug repository. The bug analyzer proposed in [17] combines GUI specifications and historical bug information with the results of image processing. However, in this work we limit the scope of the bug analyzer to use only image processing information. Integration of GUI and bug information will be addressed in a future work.

Though the focus of this paper is the bug analyzer (dotted-line components on figure 2), in the interest of completeness, we describe each of the framework components below.

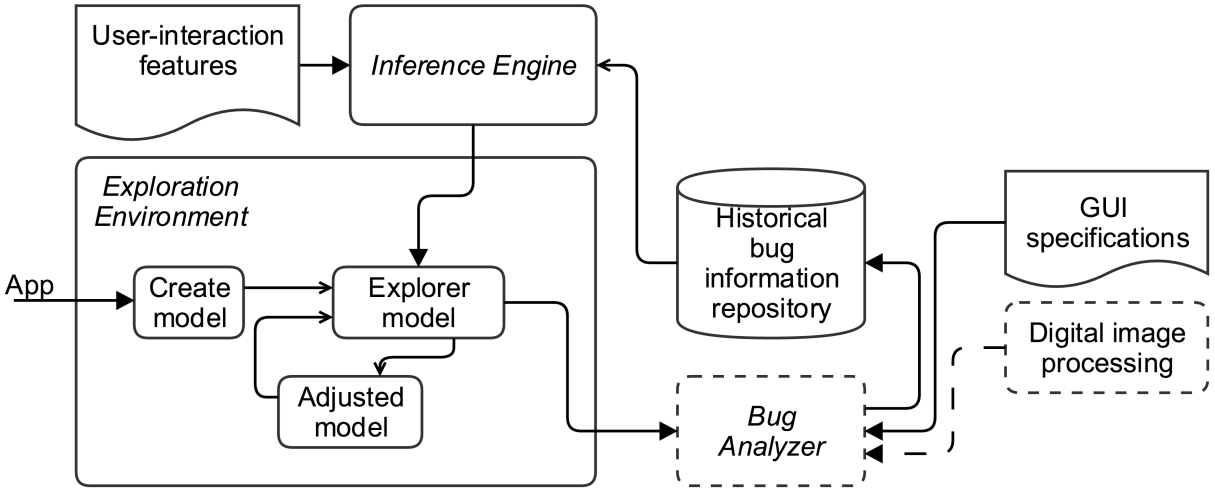

Figure 2: Overview of the automated testing framework, context of the bug analyzer.

3.1 Exploration Environment

The exploration environment is responsible for creating a model of the application. This model is used to explore the application automatically. SwiftHand [15] automatically creates a model and explores the application. We are modifying SwiftHand to allow the inclusion of user-interaction features while it explores the model of the application. This user-interaction features are included randomly. Also, we are adding the functionality to capture images and capture GUI information before and after the execution of the user-interaction features. These images and the GUI information are sent to the bug analyzer to find possible bugs in the application.

3.2 Bug Analyzer

The bug analyzer receives information on the activity being explored. Two images are captured when we apply the user-interaction feature. The first image is captured before applying a user-interaction feature being tested, and the second image is captured after applying. We used the interest point detector and descriptor SURF [18] to get interest points in each image. The interest points with the specifications of GUI are used to determine whether there are bugs produced by the user-interaction feature applied. If bugs are found, they are stored in the repository of historical bug information.

GUI information is used to identify widgets that generated the bug. GUI information is obtained before and after the applying of a user-interaction feature being tested.

3.3 Inference Engine

The inference engine is responsible for selecting the user-interaction features to be tested in the application as well as determine the order in which they will be tested. For this version of the framework, it is using a random selection of user-interaction features to be tested in applications.

Once the historical bug information is stored in the repository, the inference engine uses this information to select the user-interaction features and the order in which they will be tested in applications.

3.4 Bug Repository

The framework has a repository for storing historical bug information. In the future, the inference engine to create test cases could use historical bug information. Our inference engine, using machine learning, consults this repository and makes inferences about how and which user-interaction features to apply in new testing.

3.5 Example

We selected the user-interaction feature “Double rotation” to show the preliminary results obtained with the proposed bug analyzer. Double rotation feature expresses the act of rotating a mobile device and then rotating it back to the original orientation. With this action the application should stay in the same state.

When we perform the double rotation action, the active activity changes from resumed to paused state, meaning this activity is restarted and resumed. During this process the application may present some defects with setting data to different application resources.

3.5.1 Image Capture

The capture of two screens of an application was developed and used by us for the purpose of testing. In this case the user typed the username "Abel". We captured an image of the screen, shown in Figure 3(a). We applied the "Double rotation" feature. We again captured an image of the screen, shown in Figure 3(b). The username "Abel" was changed to the text "username". This type of bug is very common in mobile applications.

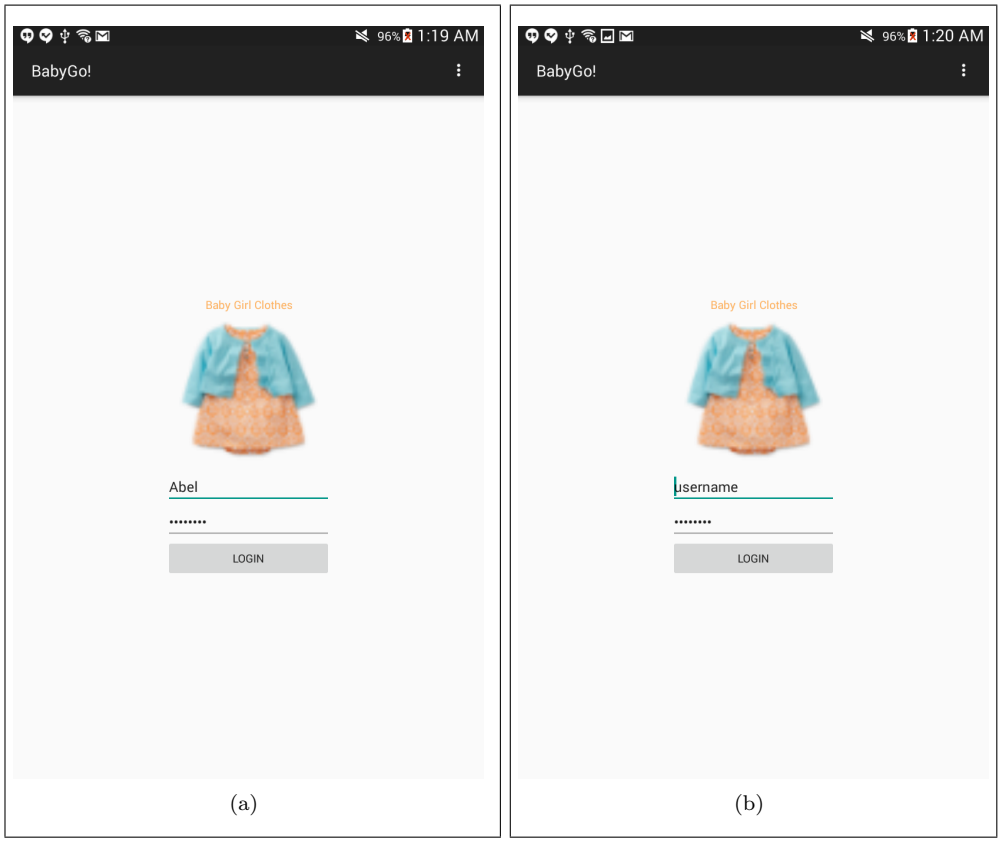

Figure 3: user-interaction feature "Double rotation".

3.5.2 Interest Point Detector and Descriptor

Interest points are places or points within the image with a strong characterization and consistent reference such as corners, edges and interceptions. The way to carry out the identification and subsequent comparison of interest points is through their descriptors. Descriptors are feature vectors calculated on each of the interest points, in other words, a descriptor is a numerical description of the image area referred to by an interest point. Particularly, SURF [18] uses a similarity threshold based matching strategy, whereby two regions are matched if the Euclidean distance between their descriptors is below a threshold.

We used the SURF algorithm included in the OpenCV library[1], version 2.4.10. This library uses a K-nearest neighbor search for obtaining the similarity of the images.

The first task was to apply SURF to the captured images of before and after the "Double rotation" feature. The red circles in the images of Figure 4(a) and Figure 4(b) represent the interest points detected by SURF. It can be seen that most of the points of interest are focused where there are alphanumeric characters. We focused on the different interest points between the two images. Different interest points between the image captured before and the image captured after we applied double rotation are shown in Figure 4(c) and Figure 4(d). We found this interest points by doing a match between all the interest points

[1] http://www.opencv.org

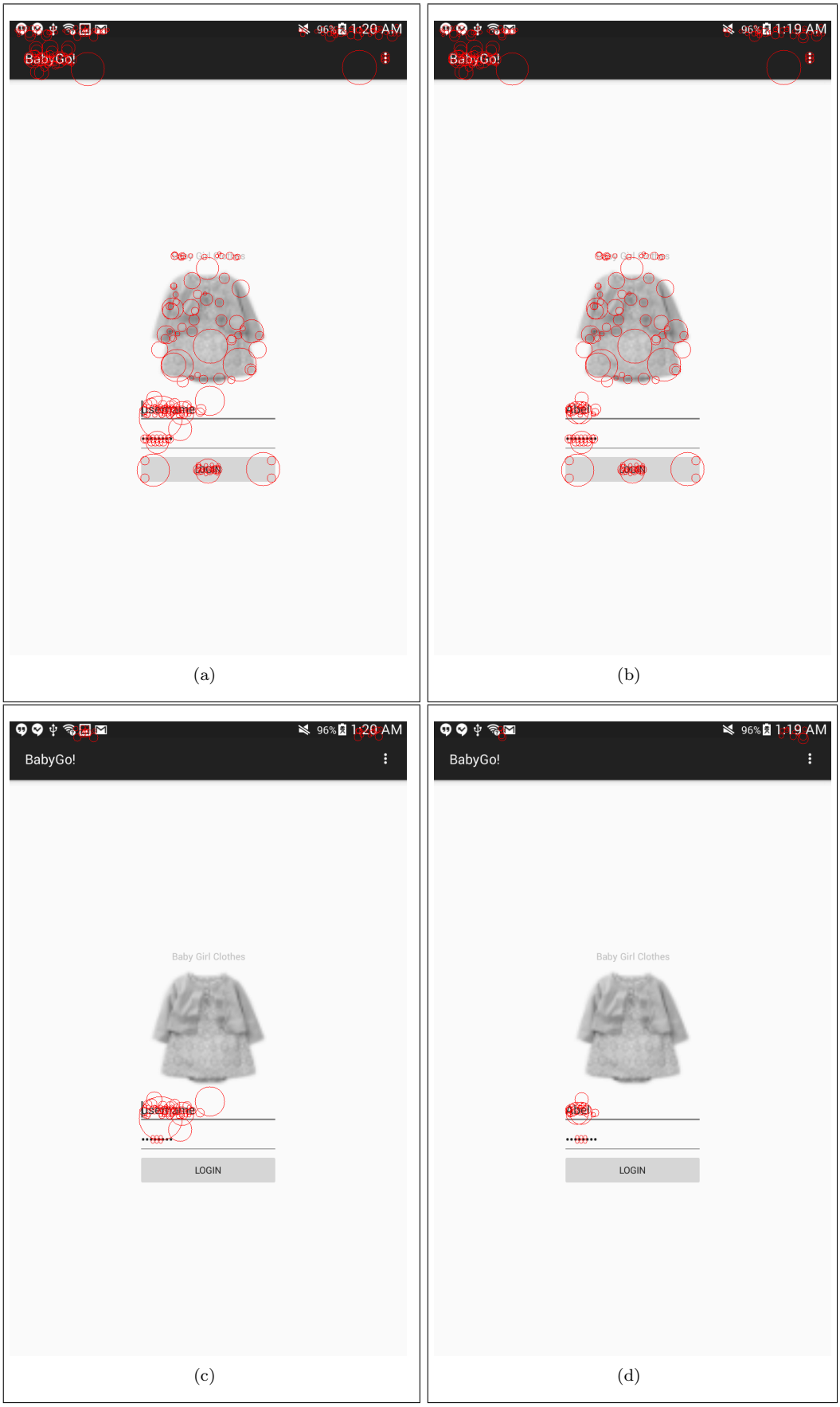

Figure 4: Interest points found with SURF.

contain on each image, and we can see the that different interest points between both images are largely concentrated on the characters of the text field.

We obtained the following data by applying the SURF algorithm: interest points in Figure 4(a) were 308, interest points in Figure 4(b) were 267, extraction time was 333.395 milliseconds, amount of similar interest points was 41, and percentage of similarity between images was 79.0941%. This value suggested important differences between the two images due to bugs in the "Double rotation" feature.

3.5.3 GUI Information

Having located the different interest points between the two images, the next step is to identify what controls are found in those coordinates to identify possible changes to these controls, given that we can obtain the absolute location (coordinates), of each one of these different interest points found on the image after the "double rotation" feature was executed. To perform this task, information from the GUI of the application should be used. With the GUI information we can determine the object type where the bugs were generated. The properties of these objects should be analyzed to determine if there were variations in their configurations.

When we obtain the different interest points from the analysis made with SURF, these interest points contain coordinates, and given that the images that we are using for analyzing the effects of the user-interaction feature have exactly the same resolution as the size of the screen of the mobile device or emulator used while exploring the app, we can use these coordinates to know exactly where in the user interface layout is located the widget that is apparently showing a bug.

To do this we obtain the absolute coordinates of each widget and the properties of width and height of each one of the widgets using the GUI structure of the application. Knowing this we can say exactly where in the screen is the widget located, so using this we can know exactly how many different interest points were found in the area where the widget is located, and decide if the amount of these is high enough to consider that this widget might be showing a bug.

On the other hand, we can, in most cases, know perfectly when a widget is showing a bug by obtaining and comparing its properties before and after the user interactions are processed. And since we have the absolute coordinates where each widget is located, we can use this to confirm that the widget considered bugged is indeed a widget that is showing some kind of bug.

4 Methodology

We conducted a case study in two phases. In the first phase we found bugs based on user-interaction features using manual testing. In the second phase we found bugs using digital imaging processing.

4.1 First Phase: Detecting Bugs Using Manual Testing

We randomly selected 15 mobile applications from the F-Droid catalogue[2]. F-Droid is an installable catalogue of free and open source software applications for the Android platform. We used manual testing to find bugs based on the user-interaction features described in section 2.2. However, out of the eight user-interaction features therein presented, only four were used and tested in this study: double rotation, killing and restarting, pausing and resuming, and back button functionality. The other four (opening and closing menus, zooming in and out, and scrolling) were out of the scope since their behavior needs to be adjusted to fit in our framework. This is a limitation of our current version of the framework. We used manual testing for approximately 20 minutes in each application and we randomly introduced user-interaction features. We captured images before and after applying each user-interaction feature. We reported if the introduction of the user-interaction feature generated a bug.

4.2 Second Phase: Detecting Bugs using Digital Imaging Processing

When we were using manual testing to find bugs, we captured an image before and after each user-interaction feature was tested. We used images to find bugs using digital imaging processing. We used SURF (interest point detector and descriptor) to find different interest points between each pair of images. Based on our experience we defined a threshold to compare the percentage similarity between each pair of images. If the similarity percentage between each pair of images was below this threshold we considered that the user-interaction feature generated a bug. Otherwise, the user-interaction feature did not generate any bugs.

[2]https://f-droid.org

5 Results and Discussion

5.1 RQ1: Based on user-interaction features, how often do bugs appear?

It is very important to clarify that we reported these bugs based on the definition and behavior expected from the user-interaction features. We selected 15 mobile applications and applied manual testing introducing different user-interaction features. Each user-interaction feature was tested independently. Each feature user-interaction feature is composed of a sequence of events. To define the sequence of events for each user-interaction, we first conducted a quick exploration of the application to understand the application logic. Then we selected the user- interaction feature to test and defined the sequence of events for it. The sequence of events included capturing an image before entering the user-interaction and capturing an image after entering the user-interaction feature. We tested 49 user-interaction features in total. Sequence of events used for each user-interaction feature are available in one external file[3].

Table 1: Comparison of manual testing and digital imaging processing.

Number	App Name	Test Id	UIF Type	Manual testing, Bug?	Digital imaging processing, Bug?	Percentage Similarity
01	aagtl	pair-000	dr	yes	yes	86.3
02	aagtl	pair-001	pr	yes	yes	76.85
03	aagtl	pair-002	kr	yes	yes	42.86
04	ApkTrack	pair-000	dr	yes	yes	43.14
05	ApkTrack	pair-001	pr	no	yes	89.86
06	betrains-nmbssncb-belgium	pair-000	dr	no	yes	93.92
07	betrains-nmbssncb-belgium	pair-001	dr	no	no	99.33
08	betrains-nmbssncb-belgium	pair-002	pr	no	no	96.05
09	betrains-nmbssncb-belgium	pair-003	kr	no	yes	93.92
10	crickets-alarm	pair-000	dr	no	no	98.96
11	crickets-alarm	pair-001	dr	yes	yes	94.54
12	crickets-alarm	pair-002	dr	yes	yes	91.08
13	defcol	pair-000	dr	no	no	100
14	defcol	pair-001	pr	yes	yes	86.13
15	droidupnp	pair-000	dr	yes	yes	27.95
16	droidupnp	pair-001	pr	yes	yes	29.15
17	droidupnp	pair-002	kr	no	no	98.02
18	e-numbers	pair-000	dr	no	no	97.56
19	e-numbers	pair-001	dr	no	no	97.1
20	e-numbers	pair-002	pr	no	no	99.61
21	e-numbers	pair-003	pr	no	no	98.23
22	e-numbers	pair-004	dr	no	yes	90.88
23	ep-mobile	pair-000	dr	yes	yes	77.85
24	ep-mobile	pair-001	dr	yes	yes	93.27
25	ep-mobile	pair-002	pr	yes	yes	53.48
25	ep-mobile	pair-003	kr	no	no	96.71
27	exalted-dicer	pair-000	dr	yes	no	95.45
28	exalted-dicer	pair-001	dr	yes	yes	91.32
29	exalted-dicer	pair-002	dr	yes	no	95.02
30	exalted-dicer	pair-003	pr	no	no	95.85
31	jlyr-lyrics	pair-000	pr	no	no	99.3
32	jlyr-lyrics	pair-001	dr	no	no	96.28
33	jlyr-lyrics	pair-002	kr	no	no	97.5
34	jlyr-lyrics	pair-003	dr	no	no	95.61
35	Screen-Notifications	pair-000	dr	yes	yes	12.04
36	Screen-Notifications	pair-001	pr	no	no	96.84
37	Screen-Notifications	pair-002	dr	no	no	99.77
38	sokoban	pair-000	dr	no	yes	72.75

Continued on next page

[3]http://eseg-cr.com/research/2015/CLEI-Extended-Tests.pdf

Table 1 – continued from previous page

Number	App Name	Test Id	UIF Type	Manual Testing, Bug?	Digital Imaging Processing, Bug?	Percentage Similarity
39	sokoban	pair-001	pr	no	yes	67.82
40	sokoban	pair-002	kr	no	no	96.0
41	Todays-bRead	pair-000	pr	no	no	95.88
42	Todays-bRead	pair-001	kr	no	yes	87.07
43	WWWJDIC-for-Android	pair-000	dr	no	no	99.75
44	WWWJDIC-for-Android	pair-001	pr	no	no	99.41
45	WWWJDIC-for-Android	pair-002	kr	yes	yes	94.01
46	yaab	pair-000	dr	no	no	99.57
47	yaab	pair-001	pr	no	no	99.04
48	yaab	pair-002	kr	no	no	99.13
49	yaab	pair-003	bb	no	no	100.0

In table 1 we reported the findings for each user-interaction feature applied to each application. In the first column we enumerated the user-interaction features. The second column shows the application name. The third column shows the test number for the application indicated in the second column. The fourth column indicates the user-interaction feature (UIF) applied in this test (abbreviations of user-interaction features were defined in Section 2.2). The fifth column indicates if manual testing reported a bug. The last two columns will be discussed in the next section.

In Table 2 we reported the bugs that we found in each application and the user-interaction features tested using manual testing. We did not report bugs that did not belong to user-interaction features. Column *Test ID* indicates the number of the test where we found a bug for the application indicated in the column *App Name*.

Table 2: Bugs found using manual testing.

Number	App Name	Test Id	User-Interaction Features
01	aagtl	pair-000	double rotation
02	aagtl	pair-001	pausing and resuming
03	aagtl	pair-002	killing and restaring
04	ApkTrack	pair-000	double rotation
05	crickets-alarm	pair-001	double rotation
06	crickets-alarm	pair-002	double rotation
07	defcol	pair-001	pausing and resuming
08	droidupnp	pair-000	double rotation
09	droidupnp	pair-001	pausing and resuming
10	ep-mobile	pair-000	double rotation
11	ep-mobile	pair-001	double rotation
12	ep-mobile	pair-002	pausing and resuming
13	exalted-dicer	pair-000	double rotation
14	exalted-dicer	pair-001	double rotation
15	exalted-dicer	pair-002	double rotation
16	Screen-Notifications	pair-000	double rotation
17	WWWJDIC-for-Android	pair-002	killing and restarting

We reported 17 bugs based on user-interaction features (Table 2). We applied manual testing to 15 applications and we found bugs based on user-interaction features in 8 different applications. We reported

11 bugs associated to double rotation, 4 bugs associated to pausing and restarting, and 2 bugs associated to killing and restarting.

5.2 RQ2: How well can digital image processing detect bugs based on user-interaction features?

We created a confusion matrix comparing manual testing vs. automated testing, for various thresholds. We captured images before and after applying the user-interaction feature. If the percentage similarity between the images was under a threshold, we reported that the user-interaction feature produced a bug. Otherwise, we did not report any bug. We used five different thresholds of percentage similarity: 87.5, 90, 92.5, 95 and 97.5. Table 3 shows the results of comparing manual vs. automated testing using five thresholds of percentage of similarity (column 1), in terms of the confusion matrix: true positive (column 2), false positive (column 3), true negative (column 4), false negative (column 5), and three other metrics: accuracy (column 6), recall (column 7), and precision (column 8). We set the percentage similarity threshold of our bug analyzer to 95%, as it shows better recall and accuracy.

In table 1 we reported the findings for each user-interaction feature applied to each application. The fifth column indicates if manual testing reported a bug. The sixth column indicates if digital image processing found a bug (threshold fixed in 95%). The last column shows the percentage of similarity between the images captured before and after we applied the user-interaction feature.

Table 3: Confusion matrix using different thresholds.

Threshold	TP	FP	TN	FN	Accuracy	Recall	Precision
87.5	10	3	29	7	0.80	0.59	0.77
90.0	10	4	28	7	0.78	0.59	0.71
92.5	12	5	27	5	0.80	0.71	0.71
95.0	15	7	25	2	0.82	0.88	0.68
97.5	17	17	15	0	0.65	1.00	0.50

We selected for our bug analyzer the threshold fixed in 95%. For this threshold accuracy = 0.82, recall = 0.88, and precision = 0.68. Digital imaging processing was able to detect 15 bugs of 17 bugs detected when we using manual testing. We used the images captured when applied manual testing and we introduced the user-interaction features.

6 Conclusion and Future Work

In this paper, we have proposed and evaluated a bug analyzer based on user-interaction features to detect bugs in mobile applications for the Android platform. Our bug analyzer uses an interest point detector and descriptor to identify new bugs. We conducted a case study. First, we identified bugs based on user-interaction features and applied manual testing to 15 applications. Second, we used digital image processing to identify bugs and we compared the results with manual testing.

Using manual testing we identified 17 bugs attributed to user-interaction features. We reported bugs in 8 out of 15 applications. Bugs based on user-interaction features are present in mobile applications and these bugs threaten the quality of the applications.

We used digital image processing to identify 15 bugs. Findings show that digital image processing is an alternative to finding bugs based on user-interaction features. The advantage of using digital image processing is that the source code is not required. Of course, we need to do more experimentation using digital image processing.

We are planning to design a new case study to evaluate digital image processing combined with GUI information to find bugs based on user-interaction features.

Acknowledgments

This research was supported by the Costa Rican Ministry of Science, Technology and Telecommunications (MICITT), as well as by the Department of Computer Science from Costa Rica Institute of Technology. Our thanks to the Empirical Software Engineering (ESE) Group at University of Costa Rica.

References

[1] H. Ham and Y. Park, "Mobile application compatibility test system design for android fragmentation," *Communications in Computer and Information Science*, vol. 257 CCIS, pp. 314–320, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27207-3_33

[2] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *ICCET 2010 - 2010 International Conference on Computer Engineering and Technology, Proceedings*, vol. 2, 2010, pp. V2297–V2301. [Online]. Available: http://dx.doi.org/10.1109/ICCET.2010.5485442

[3] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: Automated remote ui testing on android," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, MUM 2012*, 2012. [Online]. Available: http://dx.doi.org/10.1145/2406367.2406402

[4] L. Lu, Y. Hong, Y. Huang, K. Su, and Y. Yan, "Activity page based functional test automation for android application," in *Proceedings of the 2012 3rd World Congress on Software Engineering, WCSE 2012*, 2012, pp. 37–40. [Online]. Available: http://dx.doi.org/10.1109/WCSE.2012.15

[5] B. Jiang, X. Long, and X. Gao, "Mobiletest: A tool supporting automatic black box test for software on smart mobile devices," in *Proceedings - International Conference on Software Engineering*, 2007. [Online]. Available: http://dx.doi.org/10.1109/AST.2007.9

[6] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings*, 2012, pp. 29–35. [Online]. Available: http://dx.doi.org/10.1109/IWAST.2012.6228987

[7] Z. Wang, S. Elbaum, and D. Rosenblum, "Automated generation of context-aware tests," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 406–415. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.18

[8] D. Amalfitano, A. Fasolino, P. Tramontana, and N. Amatucci, "Considering context events in event-based testing of mobile applications," in *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013*, 2013, pp. 126–133. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2013.22

[9] D. Amalfitano, A. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, 2011, pp. 252–261. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.77

[10] R. Zaeem, M. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, 2014, pp. 183–192. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.31

[11] "Activities." [Online]. Available: http://developer.android.com/guide/components/activities.html

[12] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 - Proceedings*, 2013, pp. 224–234. [Online]. Available: http://dx.doi.org/10.1145/2491411.2491450

[13] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon, "Mobiguitar – a tool for automated model-based testing of mobile apps," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2014. [Online]. Available: http://dx.doi.org/10.1109/MS.2014.55

[14] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 641–660, 2013. [Online]. Available: http://dx.doi.org/10.1145/2509136.2509549

[15] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 623–639, 2013. [Online]. Available: http://dx.doi.org/10.1145/2509136.2509552

[16] W. b. Yang, M. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7793 LNCS, pp. 250–265, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37057-1_19

[17] A. Méndez-Porras, J. Alfaro-Velásco, M. Jenkins, and A. M. Porras, "Automated testing framework for mobile applications based in user-interaction features and historical bug information," in *XLI Conferencia Latinoamericana en Informática*, Arequipa-Perú, Octubre 2015. [Online]. Available: http://dx.doi.org/10.1109/CLEI.2015.7359996

[18] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision - ECCV 2006*, vol. 3951, 2006, pp. 404–417. [Online]. Available: http://dx.doi.org/10.1007/11744023_32