جامعة بيرزيت
BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Master of Software Engineering (SWEN)

Master Thesis

Automated Black Box Testing Approach for React Native App Lifecycle

Author: Ibtisal Awashrah

Supervisor: Dr.Samer Zein

Abstract

In recent years, large number of people are tending to use smartphones. These smartphones are using different OSes that need different platforms to develop their apps. These differences lead to difficulties in developing and testing the same app for the different platforms. Accordingly, the importance of cross-platforms development that produce a single app for multiple platforms is rising. React-native is an example of a cross-platform mobile app development solution. It is a pioneer single mobile app development platform, which is widely used nowadays. Because of react-native importance, it is necessary to focus on its challenges. One of the most important challenges that react-native apps are suffering from is the miss handling of the app lifecycle because it is distinct from the native apps, which is a critical problem that leads to apps crash in most cases. This paper proposes a black-box automation testing approach that tests the react-native app lifecycle, reducing the miss handling issues. The approach is an event-driven automated black-box testing framework that can explore react-native apps under test to analyse the issues. Therefore, this framework will help developers create react-native apps with a minimum number of crashes that cause stopping app functionalities with unexpected exit, and GUI errors caused by the lifecycle. The framework addresses lifecycle key loops of the react-native using double orientation activity or background foreground activity. The framework has been evaluated by checking its ability, to detect the apps crashes, and the different states of GUI positions, on an open-source application provided by MIT with the issues injected in. The final output from the framework is JSON log file with properties or disappearance caused by miss handling of lifecycle. The main results of the framework evaluation found that the framework detects 60% of the apps crashes and 100% of GUI errors caused by lifecycle mishandling.

Acknowledgement

First and foremost, I am extremely grateful to my supervisor Dr Samer Zein for his invaluable advice, continuous support, and patience during my thesis research. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research. Also, I would like to express my gratitude to my parents, my sister and my brothers. Without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete my study.

Contents

List of Figures

1 Introduction

According to studies, there are more than 3.8 billion users of smartphones in 2021 [1]. Google Android and Apple iOS have been the two dominant mobile platforms in the last two years [2]. Because mobile users rely on mobile applications to pass their daily tasks, successful mobile apps should target these two platforms to maintain significant user coverage. However, developing two different native apps to target each platform consumes a lot of time and cost. Recently, several cross-platform frameworks have enabled developers to build a single app base code and deploy it on both platforms. Successful application development became possible using cross-platform. One of the pioneers cross-platform is react-native, powered by the Facebook team, react-native based on JavaScript to form a one-code base for both IOS and Android possible [3].

Cross-platform increase the number of provided applications for variant uses. Therefore, mobile application users become highly dependent on mobile applications for daily and critical domains, the demand for high-quality mobile applications has grown. Thus, the mobile developer's goal is to give quality proper consideration. As a consequence, the developers should be adapting suitable quality assurance techniques and testing. Mobile developers could use many mobile testing techniques to increase the application quality, such as automated testing using tools that focus on automating the tests routines and repeat. For the react-native applications, the developers could face many challenges such as handle the lifecycle [4] of the applications based on react-native.

Many works of literature have discussed mobile applications suffering from lifecycle mishandling, such as [5] [6] [7]. Zein et al. [8] performed a systematic mapping study for mobile application testing techniques using 79 papers with identifying many possible areas that could be handled in future works. One of the main future works was targeting the activity of the mobile application lifecycle.

The main solutions of lifecycle mishandling issues in the literature proposed testing techniques that depend on Graphical User Interface (GUI) models or testing artefacts that generate automatic test cases for the activity lifecycle. In [7] detected bugs that may cause applications to be stopped, paused or killed; their solution could generate automatic test cases. Other solutions suggest dynamic analysis for finding issues using specific types of failures because of Android system failure and emulator crashes[9]. However, all these suggestions were for Android applications.

This research proposes our solution for applications based on a react-native framework (React-Native Lifecycle Ripper). Our framework fully automated event-based black-box dynamic testing technique. Our framework detects application crashes and GUI errors. Specifically, our framework tests GUI properties change (CSS, props, and HTML tags), GUI disappearance of existing objects, and GUI adding new objects related to lifecycle activity using systematic testing each GUI activity state encountered during application exploration. To reach our goal, the framework tests the lifecycle of react-native; it is designed to test oracles[10][11]. Oracles is a testing mechanism that determines if the system's output is the same as predicted from system design [10].

After generating oracles, one of the main events Double Orientation Change event(DOC), Background Foreground event (BF) [11], will be activated to hit lifecycle key loops. Our framework will generate the new GUI results and compare them with the main oracles. This allows our framework to generate the specific types of failures (GUI errors and app crashes) that could be caused by mishandling of lifecycle.

To measure our framework effectiveness, we injected four types of bugs in the open-source application provided by MIT. We picked an open-source application that has already been tested and licensed from MIT; this guarantees the bugs' detected by the framework is from the injected bugs, also this methodology was used in other papers such as [12], where the authors evaluated their work using the same evaluation method. We have injected one failure for each testing cycle. Our framework detected each failure that we added in each testing application. Our framework is the first approach is designed for catching lifecycle mishandling in the react-native application.

1.1 Research Motivation

React-native developers should consider the lifecycle of their application. Thus, they have to implement the lifecycle components correctly. Therefore, the users will not expect aberrant transactions during different lifecycle transitions of the react-native applications. However, many unexpected behaviours could occur; the main unexpected behaviours that the framework focuses on testing in the mobile applications based on are GUI errors [5] and crashes[6], more details of the failures could be found in chapter 5.

- **Crashes Failures:** this type of failure occurs when an unexpected stop of the mobile application functionalities, the unexpected behaviour that we will focus on in our framework is the unhandled exceptions that occur because of the mishandling of the lifecycle, this cause termination of the process using the Android native application functionalities.
- **GUI Errors:** this is relevant to lifecycle key loops, specifically Update key Loop and Entire Loop that we will explain in the background chapter. Our framework checks GUI errors caused
  - As wrong positions from the original GUI positions[5]
  - As wrong object properties (props and CSS style or HTML type)
  - Object disappears

This research aims to create an online black-box automation framework; our framework will handle the lifecycle testing of react-native applications in Android OS. Our research is depending on the ALARic methodology[11]. ALARic is ”fully automated Black-Box Eventbased testing technique that explores an application under test for detecting issues tied to the Android Activity lifecycle”. ALARic methodology is specified for the online testing of Android native applications.

However, our framework is more specified for react-native applications for Android OS. Thus, we have a framework that can handle the full lifecycle of applications based on react-native. The full lifecycle of the react-native (Mounting, Update, Unmount) and native lifecycle of the OS that the application running on.

Furthermore, our research has a notable contribution to the literature on automated react-native GUI testing:

- GUI testing technique to detect app crashes and GUI errors tied to the react-native Activity lifecycle. Furthermore, our framework is the first dynamic testing technique that can address the issue of GUI errors and app crashes resulted from life-cycle for react-native mobile applications;

1.2 Research Questions

The main goal of our research is to create a solution approach to detect background action crashes that are related to the react-native lifecycle. In our research questions, we focus on the errors that we discussed in the previous section after triggering the main key loops that we will explain in the background chapter.

- **RQ1:** How effective is our framework in detecting crashes in the applications that are based on the react-native activity lifecycle?
- **RQ2:** How effective is our framework in detecting GUI errors of changing in GUI object positions, change object properties (props, CSS, HTML), disappear objects that are based on the react-native activity lifecycle?

1.3 Structure of the Thesis

In this research, we followed this structure. For Chapter 2, we will describe the background of react-native. Chapter 3 has a literature review for related works. Chapter 4 will present our methodology. Chapter 5 will describe how we performed the evaluation of our framework; Chapter 6 will have our experiment results. Finally, Chapter 7 will provide the conclusion and future work.

2 Background

2.1 React-Native

React-native ”is a framework based on JavaScript; it renders mobile application for both iOS and Android natively. React-Native developed by Facebook ” [13]. React-native in 2015 starts to support iOS in its mobile application. Then because of the active community of the Android community, which supports adding many other contributions to the open-source project; react-native start supporting Android OS[3]. This framework allowed for the development of more than 7000 new application during 2019-2020 [14]. React-native uses the fundamental blocks from the native platform to build the UI, UIView on iOS, or View on Android. Thus, the developers could build their applications using native components but with higher-level language. Furthermore, react-native uses threads utilization to run mobile applications with the necessary operations[15]. The main two threads that are doing react-native work are the JavaScript thread written using JavaScript that is responsible for the business logic application running. The JavaScript thread mainly sends/receive requests to/from the main thread to draw views and respond to events. According to JavaScript thread requests, the second thread is the main thread responsible for drawing UI and responding events such as touch events, network requests, Etc.[15]. These requests that are sent from react-native internally during components are called bridges.

The main two threads that are doing react-native work are the JavaScript thread written using JavaScript that is responsible for the business logic application running. The JavaScript thread mainly sends/receive requests to/from the main thread to draw views and respond to events. According to JavaScript thread requests, the second thread is the main thread responsible for drawing UI and responding events such as touch events, network requests, Etc. [9]. These requests that are sent from react-native internally during components are called bridges. React-native application GUI uses VDOM (Virtual Document

Object Model), which allows to dynamic access of HTML tags by generating all tags as object tree model[16].

For running JavaScript recently react-native developers are using expo SDK [17]. Expo is an open-source platform working to enable and runs react-native applications. Expo has a contribution in simplify react-native application testing [17].We use expo SDK according to easy installing and lunching of expo projects.

Using expo the react-native application apk/aab can be built in expo cloud, the source code is uploaded and the expo cloud will provide the final bundle which could be downloaded and runs in the needed device [18]. Unlike react-native CLI, in which all the building processes should be handled in the system, this usually causes build errors such as Gradle" which is a build automation tool for multi-language software development" failure[18], manifest errors, this type of errors causes a direct effect in the react-native application build. Our framework was developed using react-native based on expo which has a great usage of the developers who need to use this framework for easy lunching of the framework, without any complicated setup. In addition, the expo offers a collection of solutions that helps our framework execute the events such as Back Handler; Back Handler allows the application to be exited. In addition to the Linking solution which allows the application to be relaunched, these libraries are not found directly in react-native CLI.[19].

2.2 React-Native Lifecycle

For react-native applications that use native building blocks, the activities are essential. For example, for Android Framework, the activities are a subclass of the Activity class, which manages the user exercises using the Activity stack. The activity stack allows the user to navigate several screens by putting the foreground screen at the top of the stack. However, for the react-native, the Activity lifecycle is pretty different. The react-native has two lifecycles. The first lifecycle is the native lifecycle UIViewController on iOS and the activity lifecycle for Android [15]. As our research specifies react-native in Android OS, we will explain more about the activity lifecycle. Mainly Android OS deals with the applications activity as the stack; that means that the application will be running in the OS memory storage; this allows Android to manage the applications. Therefore, the application will have four primary states:

- **Foreground State:** Application in the foreground state is at the top of the stack, it will be visible.
- **Visible State:** Application in the visible state is visible to the user, but it lost the focus, such as the application is non-full size.
- **Stop State:** Application in the stop state, when the application is stopped or hidden the background.
- **Destroy State:** Application in the destroy state process is killed.

To Reach these states the application will pass these phases [20]:

- **On Create:** In this phase, the activity will be created, standard static setup will be had like binding data list, this state will be frozen until the activity goes to on Start .
- **On Start:** This state when the application becomes visible to the user.
- **On Resume:** this state called when the application is in the foreground to interact with the application
- **On Pause:** the application loss foreground state.
- **On Stop:** the application no longer visible to the user.

- **On Destroy:** kill the application process.

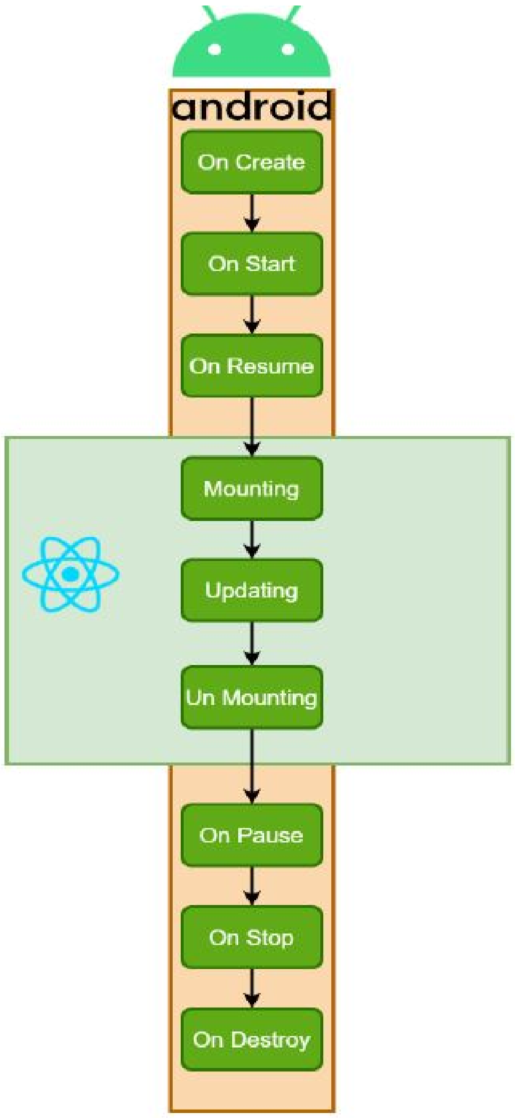


Figure 1: Diagram presenting the phases of the RN Component lifecycle

As shown in Figure 1, the second phase is separated into three steps: The first step is Mounting: it is done once per react-native wrapping view's lifespan. After calling the componentDidMount, the component is bound with the native view as first render using initial props and state [15]. The second step is the Update phase: the components run through this phase with every new pops or state has changed or received. The final step is the Unmount phase which responsible in detaching the component from the native view[15].

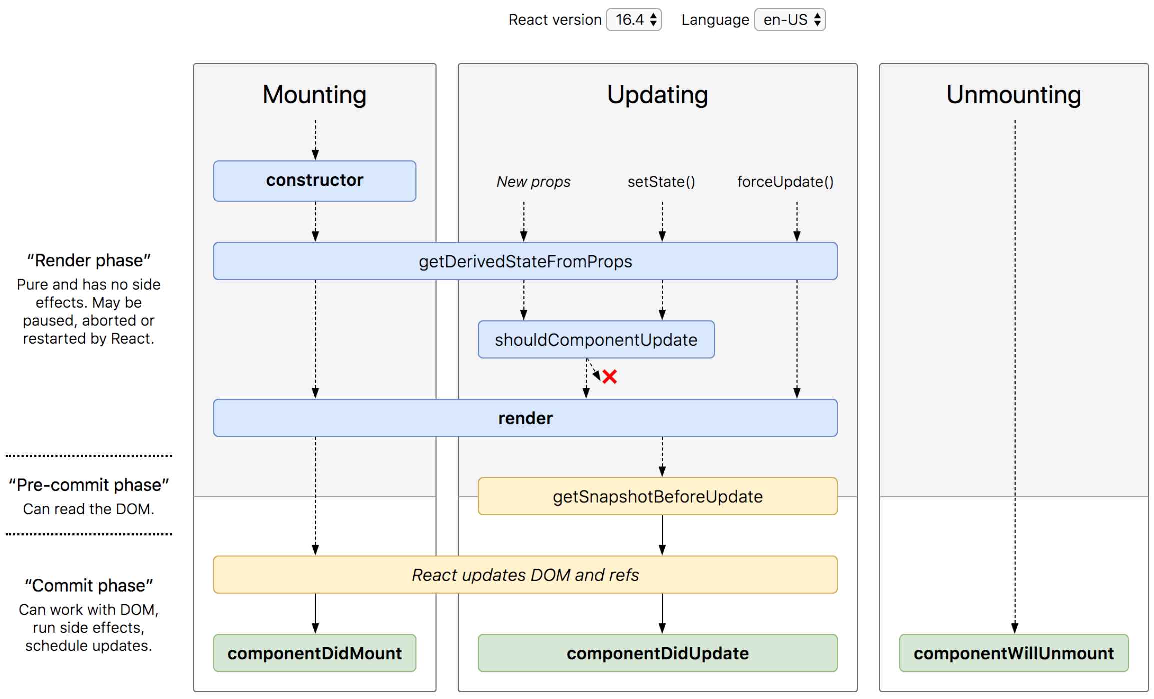


Figure 2: Diagram showing the detailed lifecycle of React.Component [20]

There are four key loops of the Activity lifecycle. The main key activities for detecting the main four key loops are according to React-Native Guide and Android Guide. Thus, these loops are Entire Loop, Visible Loop, and Foreground Loop and Update Loop are at first, The Entire Loop (EL) of an Activity consists of the Resumed-Mount-update-unMount- Paused-StoppedDestroyed-Created-Started-Resumed sequence of states. This loop can be exercised by events that cause a configuration change, such as an orientation change of the screen, that destroys the Activity instance and recreates the application according to the new configuration. Secondly, **Visible Loop (VL)** corresponds to the Resumed-Mount-update-unMount-Paused-Stopped-Started-Resumed sequence of states Activity is hidden and then made visible again. There are several event sequences able to stop and restart an Activity, such as turning off and, on the screen, or putting the app in the background and then in the foreground again through the Overview or Home buttons.

The third loop is **Foreground Loop (FL)**of Activity involves the Resumed-mount-update-unmount-Paused-Resumed state sequence. The Resumed-Paused transition could have happened when non-full-sized elements are opening, for example, semi-transparent of the activities or model dialogue. This transition occupies the foreground while the Activity is visible in the background. So, the users should discard this element when it triggers the transition of Paused-Resumed. To trigger the transition Paused-Resumed, the user should discard this element.

The fourth loop is **Update Loop (UL)** of Activity involves the update - update state sequence. This can be triggered by updating UI and change UI elements

3 Related Work

3.1 Introduction

Smartphones have been circulated through most people's lives during the last ten years. It has been making their user's lives easier using different applications that specialized in covering their daily needs. These needs could be covering communication, arrange needs, Etc.

From these benefits, it was essential to arrange these application developments in different OSes using cross-platform development. In addition to checking increasing applications performance and decreasing applications' errors using mobile application testing. One of the primary checking topics in this research is lifecycle testing for the cross-platform application.

This chapter presents a comprehensive review for previous studies that have approached mobile application testing's, mobile application development using cross-platform and previous tools developed for mobile application testing.

3.2 Research Method

According to these criteria, we collected the related papers publishing years, database, number of pages, type of paper, and keywords. According to

- The first criterion, we collected the papers from recent years 2016-2020 because there is rapid growth in mobile applications testing studies in cross-platform technologies.
- The second criterion, we used the IEEE database, Research Gate, and Google Scholar.
- For the third criterion, all the papers we searched about have at least five pages with two columns.
- The fourth criterion: all papers should be empirical studies.
- Finally, we used many keywords for grouping the related work under a set of sections, the main keywords are mobile application testing, cross-platform applications testing, react-native vs other cross-platforms, react-native testing, mobile application lifecycle, and react-native lifecycle.

3.3 Testing of Mobile Application

The key step for a successful mobile application is to ensure application quality [21]. Testing is the most important step for application quality assurance[22]. Therefore, mobile application testing used for detecting the expected errors of the application, which helps to develop bug-free applications. In addition, testing helps to make sure that the application meeting user needs [23].

3.3.1 General Mobile Application Testing Metrics

During our literature review, we found many papers that highlighted mobile application testing; one of these papers was [8]. In [8] the authors provide a systematic mapping study of 79 papers discussing mobile testing techniques. Their study is a comprehensive study; it focused on mobile application testing from many sides, such as life cycle conformance. Their study method had five steps from define research questions, conduct the search, screen papers, keyword abstracts, also extract the data. From their study, we could conclude that there is a real gap in studying mobile application life cycle testing, from 79 papers, there are only two papers that searched about this topic. These two studies focused on the importance of the life cycle model in order to produce dependable mobile applications and services. Therefore, our study will focus on mobile application life cycle testing.

[23] defined mobile application testing matrix and case study challenges. First of all, they defined what the best mobile application for the case study is then they defined the testing matrix of mobile application which were, test scope, test level then they defined test techniques for each level according to each environment. As we are planning to do in our study, they explain each step of the case study test process. Therefore, their study managed

to answer different testing matrices and techniques, also they applied their study on the mobile application by Monkey[24]. Monkey is online testing tool that create automated testing for Android applications. However, their study explained the mattresses from the agility testing view[24]. Unlike [25] which introduces the testing matrix according to the lean canvas test strategy view.

[25] uses lean canvas board for mobile application testing, in their matrix, they define the scope, out of scope, tools, risk, resources, people, and main functionality testing. Mainly, the lean canvas is one light whiteboard that helps to divide big components, into small components, which affects saving time and reduces cost.

[26] compares between mobile application testing and web application testing, the main points that they focused on are testing types and testing tools, for the mobile application they pointed out these types of testing performance testing, usability testing and functional testing, for our framework, we will focus on application performance. Also, they pointed out the main tools commonly used for mobile application testing, such as the monkey tool which is ”to test devices and applications at different levels like function or framework and for the unit test cases, but it can also be used for other purposes”.

In [27] same as [23] the authors focused on mobile application challenges. According to [27] , the main challenges of mobile application testing considered many factors such as device hardware, screen size, connectivity issues and platform. In our framework, we are focusing on testing the application in the full lifecycle to which keeping the connectivity of the application. In addition, they went through a mobile application cycle that considering key points like environment, the application under test, which depending on the requirement needs and starting the testing from the beginning of the development.

3.3.2 Testing of Cross-Platform Applications

Cross-platform is developing apps stand out according to their ability to run in various operating systems (OSes), such as Android, iOS. [28] introduces an overview of the main tools available currently for cross-platform mobile application testing. They mainly develop an application using react-native, which is a cross-platform library, then they used Appium [28] to test their application which approved Appium effect in mobile application functionalities testing, and layout testing.

However, [29] developed a prototype a tool called x-PATeSCO. This tool main aim is to support the layout testing, as well as the eight (absolute path, ancestor index, expressions in order, expressions multi locator, (ancestor attributes, element type, cross-platform, and identify attributes) locating strategies considered. The evaluation of their approach was in 9 mobile application uses cross-platform, and they were comparing the locating strategies using six real devices. They reached 14.2 % executing time for this prototype, with 75% identifies attributes test cases, also, for identifying attributes 8.9% and absolute path for event executability 17.6%.

[21] identifies various types of testing that gave to a mobile application which developed using react-native cross-platform, their main contribution was how they could investigate to conduct all types of testing; then they implement many examples to the test facilitate the full testing. This study only studies that we find that related to all testing types for react-native applications, their study results of using multiple testing tools that web driver testing is flaky because of time precise, and variety in the performance. However, when they used Appium[21], they found that for they can use the end-to-end UI testing for mobile application and web applications with minimal changes because Appium strongly depends on Selenium[21], that could help in a variety of the performance.

3.4 React Native in Comparison with Other Platforms

Because of the mobile market's diversity, a considerable difficulty appeared to support the applications in all these devices in the same efficacy according to the GUI, lifecycle handling, etc. These devices are with multiple OSes, react-native as an example of cross-platform single support application for multiple OSes. In this section, we will present previous papers interested in comparing the react-native framework with others so that we can have positive and negative points of react-native.

In [30]they evaluated the current frameworks for cross-platform in comparison. They depend on the applications that reach the highest number of users, how much the application was consuming the energy, and how it impacts the energy consumption with the deployment identities. Therefore, the main contribution of their paper analyzed energy consumption. The results found that MoSync that uses JavaScript consumes more energy, but C++ has the worst performance to retrieve the data. So, the JavaScript solution is the fittest as a native solution; this result leads us to [17] study.

The main goal of [17] is to determine if the cross-platform is viable for modern mobile applications in two paths, the development path, and the user experience path. They used react-native applications as cross-platform applications. Their comparison results looked and feel for react-native applications seem to native applications, but they emphasize no definite conclusion because of lack of studies in this field. Also, they found react-native application easier for mentality because the same code works in both operating systems, iOS, and Android. On the other hand, there is a problem with react-native projects because it does not rely on Apple and Google SDKs updates, but on third-party organizations, mostly from the open-source community.

The same benefits were found [21] if the react-native performance is equivalent to Android applications performance, they study the problems they faced with react-native application performance. Therefore, they tried to provide insights about react-native performance according to application lunchtimes, rendering latency of components, navigation actions, and list scrolling. They found that react-native does incur a performance impact, especially on the older hardware. However, these results eliminated in the newer hardware.

In [31], the main goal was to evaluate the react-native compared with native applications. The authors were developed one application using react-native as well they developed the same application in the native version for Android and iOS platforms. According to their results, they recommend using react-native in future applications. They found that react-native could be developed in 75% of the applications, but they indicate potential CPU usage problems.

In [32] react-native has been evaluated and compared to native applications and Xamarin [32]. The authors used a quantitative, qualitative method, user testing, performance testing, and case study. They have developed the front-end part for Android and iOS applications. Their study's main result was that the react-native could be potentially be used successfully for applications development. For the bubble sort application result, react-native could not scale as a native application, not Xamarin. Finally, they found that the main problem of react-native was that it depends on the open-source community.

3.5 Mobile Application Lifecycle Testing

Much of literature and industrial work related to mobile application indicates that there are suffering from many problems because of the mishandling of activity-lifecycle development and testing[8]. Missing activity-lifecycle testing causes many crashes, memory leaks, and data losses. In this section[11], we will handle some suggested testing tools that test these points in mobile applications.

[11] the authors suggests a tool called ALARic. ALARic is a fully black-box automated testing that handles android lifecycle testing. Their tool works in the application under test (AUT); it sends random input events to AUT with a systematic execution of these inputs. Their study results found that ALARic has high effectiveness in detecting lifecycle issues[11]; it detects all lifecycle issues in 15 Android application that they experimented with. Besides, ALARic founds GUI failures and crashes caused by Android lifecycle mishandling.

In [33] they developed a tool called Data Loss Detector (DLD), this tool methodology works as ALARic which reaches the faulty activities, but it is specialized as a detector for the data loss in case of an external interruption in the running application. DLD has outperformed than ALARic in Quantum.

Unlike ALARic used for lifecycle black-box testing, in [34] they developed a tool called SAALC which could be analyzing Android life cycle activities and extracting important information about application lifecycle, using callback methods usage. These data could avoid application memory leaks, data loss, and resource compromise. They analyze more than 842 Android opensource applications that contain more than 5577 activities. The results show which corrects and the incorrect callback methods that developers usually used in their applications.

Unlike the previous tools that depend on the applications under test AUT, in [35] tools called ALCI, their automated approach is to analyze static code during different stages of the application lifecycle. Their method targets the encapsulated lifecycle system rules and they created a repository for their resources. Finally, in the novel code analysis algorithm, they fined two code patterns that most developers, also the special design patterns could make the code loosely coupled. Therefore, this tool could find the correct and incorrect releases of the code, in a good performance.

3.6 Summary

In this chapter, we handled some literature reviews of mobile application testing. Thus, we found some matrices of mobile application testing. Then we specified our review for cross-platform mobile application testing. We found some tools that handheld cross-platform testing, such as x-PATeSCO. Then we have a review for comparison between cross-platforms and react-native and why could react-native be preferable for many products. Finally, we searched for mobile application lifecycle testing. According to lifecycle testing, there are many kinds of research in these topics for Android OS such as ALARic and DLD. However, we did not find any related searches for react-native lifecycle and react-native lifecycle testing. Therefore, our research will open new future works according to this topic.

4 Methodology

In this chapter, we will present the research approach of our framework; then we will move on with our research methodology step by step with details.

4.1 Research Framework

Our framework performs an automated online testing technique and explores the application under testing AUT. It detects odd behaviours of the application lifecycle, specifically GUI errors and crashes.

Our framework detects application crashes when the react-native application stops functionality and unexpectedly exits. It also detects lifecycle GUI errors after any key loop; these errors include changes in objects' position and object properties (CSS, HTML, props) and object disappearance.

Our framework sends random requests to the Application Under Test (AUT). These requests are input events that are systematically executed. It can practice the four key activities of the react-native lifecycle, as shown in figure 3. These events sequence can trigger at least one of the key loops defined in the below figure called Lifecycle Event Sequence. During the mobile application exploration, Life Event Sequence explores a new GUI for the first time. Then our framework discovers the related issue of the react-native lifecycle. Our framework is using the primary Event Lifecycle Sequences, which are Double Oriented Change (DOC), the Semi-Transparent Intent (STAI) and Background Foreground (BF).

In Double Orientation Change (DOC), the main goal is to practice the Update loop's event sequence two times because a single event cannot detect the GUI errors correctly and satisfactorily. DOC consists of two sequential improvements of events changes; this detects the secondary changes and differences of the GUI views and contents. In other words, the first request result should be the same GUI in the second request (without any changes in JSON output). This event is for practising the Entire loop.

Semi-Transparent Activity Intent (STAI) puts the mobile application in Foreground Loop. The event is starting from foreground activity to semi-transparent activity. This activity causes pause action of the lifecycle.

Regarding practising the Visible loop, we have to lunch the Background Foreground sequence. Mainly in the mobile application, this can be handled using the home button view on the mobile.

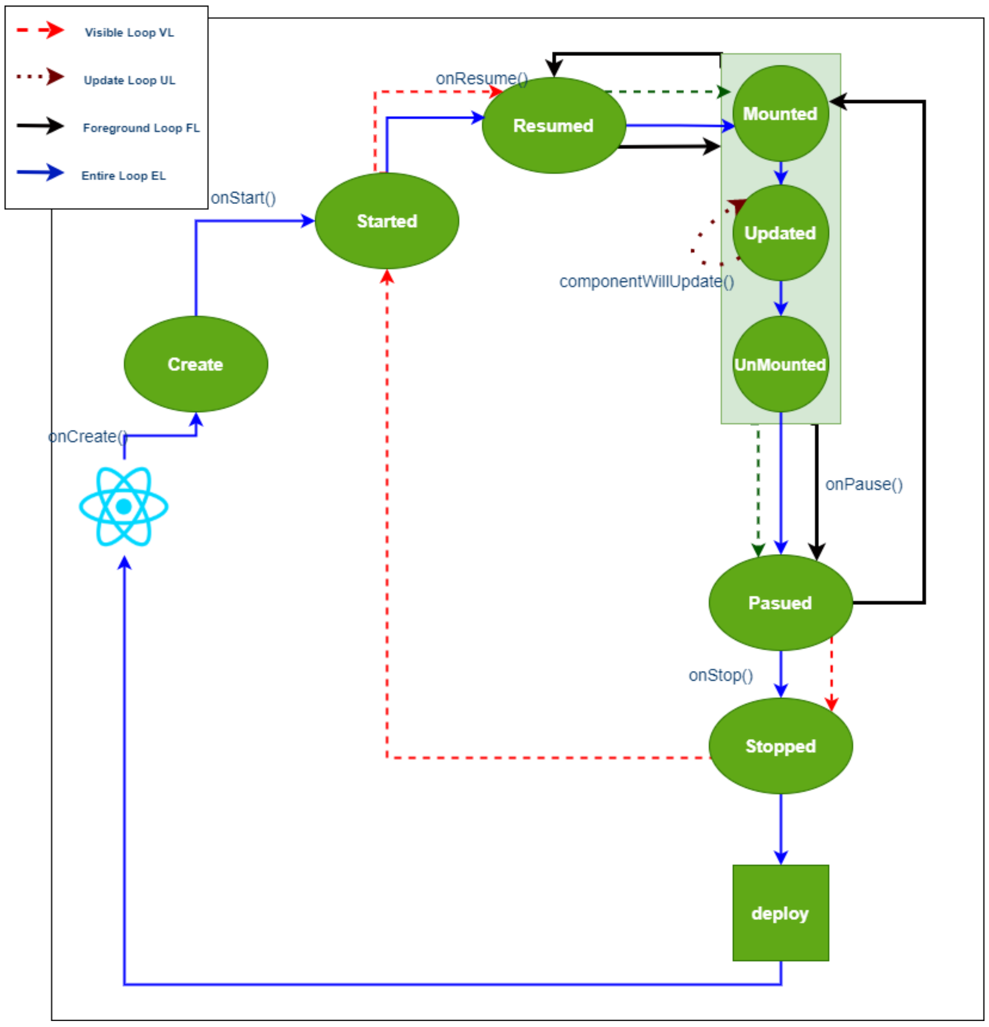


Figure 3: The React-Native Lifecycle Key Loops

4.2 Research Methodology

This section presents our research methodology steps that we work on during the two semesters.

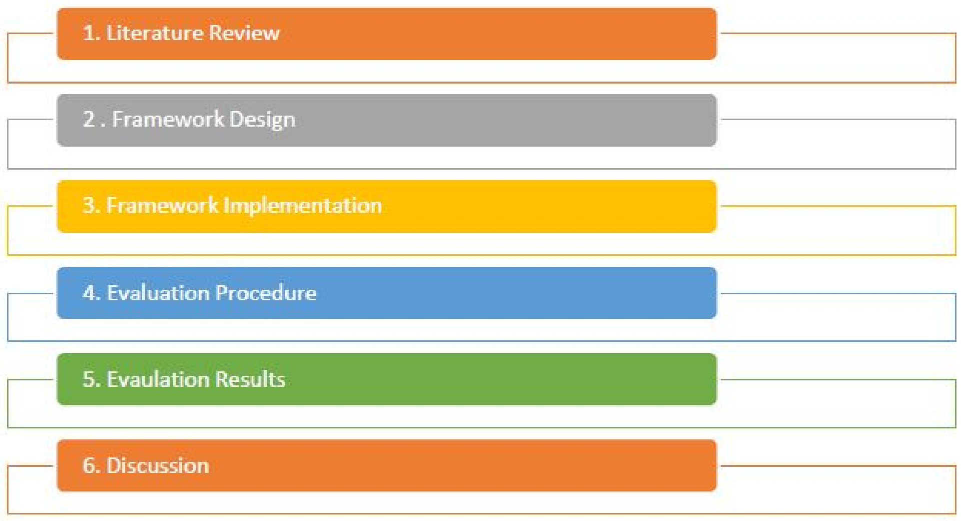


Figure 4: Research Methodology

4.3 Framework Design

Framework research implementation in our framework with the architecture shown in the following Figure 5. Our framework has two main components, the first component is the **Framework Engine**, and the second is the **Test Executor**.

The **Framework Engine** works for the testing business logic approach. Therefore, it will be responsible for GUI analysis and comparing GUI's rendered after AUT testing by using GUI attributes fetching and widgets composing as JSON output. Besides, it will not be directly responsible for input events. Thus, **Test Executor** will be responsible for the next request according to an exact plan for failure checking.

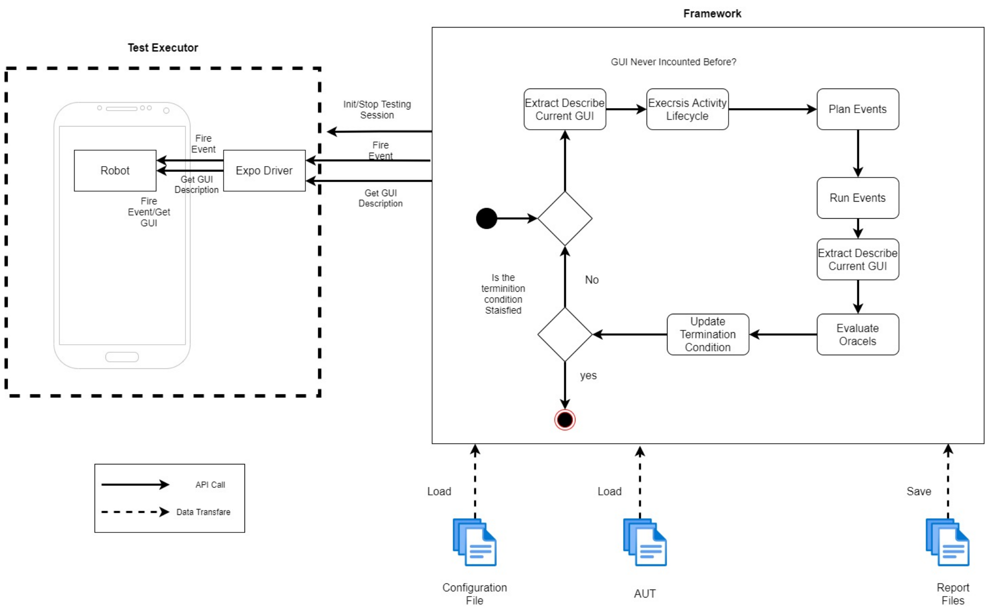


Figure 5: Framework Architecture

4.4 Framework Implementation

4.4.1 Test Executor

The react-native mobile application developer installs our framework libraries using npm. The developer adds the LifecycleStressTesting module with adding configuration of execution time or the number of cycles. For full execution, as described in figure 5; our framework will generate a random option of the events: Double Orientation Change, Semi-Transparent Activity, or Background Foreground (DOC, STAT, BF) to be executed for running the main four key loops.

4.4.2 Framework Engine

To start execution, our framework uses a test renderer library for extracting the current chosen component DOM with all properties (Objects, CSS, props) as a JSON output.

Secondly, our framework executes chosen key loop; for the DOC event, the chosen component will be double oriented between landscape orientation and portrait orientation. We use ScreenOrientation from expo-screen-orientation library to control the orientation of the application.

For the BF event, the framework puts the application in the background then recall the application again. To implement this component, we use BackHandler and Linking from expo react-native library.

For the second phase, we implement each event as a component; this allows react-native developers to test one exact key loop they are targeting. For implementing DOC, we used UNSAFE_componentWillUnmount, which was created at the start of any new application. As we mentioned before, we added an event listener that applied when the lifecycle changed in this component. In this case, we saved the VDOM JSON as an oracle point to compare any changes within it. After that, we apply double orientation for the testing phase, and we catch the changes after componentDidMount using the JSON Diff tool. This tool helps us determine the difference between the original GUI JSON and the extracted GUI JSON after applying the event. We extracted the JSON of the GUI using the renderer library from react-test-renderer. Then we save the original JSON as oracles mainly to define the main GUI properties.

The renderer library is used by the Jest framework. Jest framework is used for testing all javascript applications. The main feature that has some familiarity with our framework is the snapshot feature. By using this feature, developers can define the difference between two GUI's. Finally, for implementing crash catching, we used componentDidCatch to catch the error and resave the error type, allowing our framework to log the activity and error reason for stopping the application.

For implementing BF, like DOC, we used UNSAFE_componentWillUnmount for the initial point for saving VDOM JSON; then we applied back handler to move the application to the background. Finally, we call the application again to save the new VDOM JSON values and compare the differences between all these values, and the same as BF, we used componentDidCatch for crash handling.

We have some limitations for the STAT event component according to non-full size (visible loop) for the application in react-native. Until now, we cannot have control over this event. To control this limitation, we create STAT analysis component. This component detects the GUI oracle and handles the errors and crashes using componentWillUpdate, componentDidCatch in sequence, and application crashes using manual reaching to this activity by the developer.

Thirdly, our framework extracts GUI DOM properties as JSON (attribute, value) pairs output; then, it will compare the GUI properties after execution with the oracle JSON GUI.

Finally, our framework repeats the process of GUI exploration until it reaches the termination condition. Finally, the framework can detect GUI errors and crashes after the Event Lifecycle Sequences set and send all results to be extracted as log files.

These errors will be raised in Report File that our framework will produce. The Report File will have this structure for **crashes**:

- The mobile application names.
- The sequence of event lifecycle that causes crash.

- The name of detected failure with the activity name.
- The unhandled exception and the crash stack trace.

For the **GUI errors**, it will have this structure:

- The mobile application names.
- The sequence of event lifecycle causes GUI error.
- JSON screenshots of the failure before, and after GUI error.

As we described, our framework has a full execution as stress testing for the application. Stress testing is based on triggering the test events many times until it tests the application's performance. However, if react-native developers want to test each key loop separately, they can use the BF and DOC components. When the react-native developer uses any of these components, it detects GUI errors and crashes cased by the lifecycle mishandling.

5 Evaluation

5.1 Evaluation Procedure

We followed the evaluation procedure by two steps; the **first step** is to inject bugs into an open-source application. The main criteria that we used to select the application are the availability of the code on GitHub, a new application and maintainable code; we extracted these criteria from previous studies [11][34].
We found Expo Crud Board [1] open-source application that matched our criteria; it is a simple react-native application licensed by MIT. We tested Expo Crud Board using the BF, STAT and DOC events, and there were no bugs. Figure 6 shows the application before any bug injection. This application has 4 main components: Back component, Main Page component, Login component, and Board Button component. The **second step** is to run our framework to evaluate the execution.

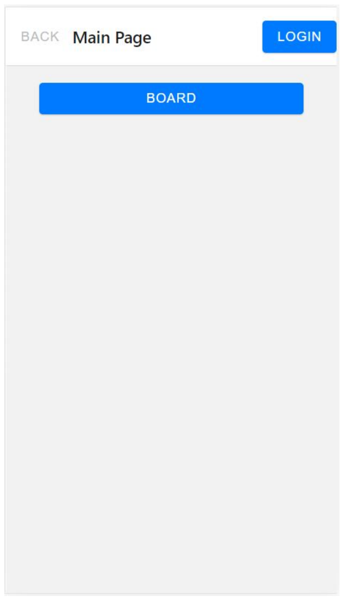


Figure 6: Expo Crud Board Application

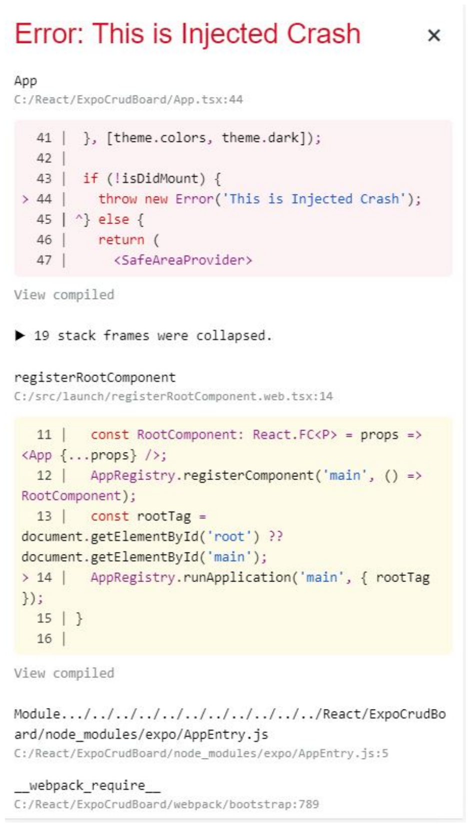
[1] https://github.com/DPS0340/ExpoCrudBoard

5.2 Bug Injection to the Application

5.2.1 Crash Failure Injection

Our target is to detect unexpected crashes of react-native applications. We injected errors into the open-source application by raising errors randomly after touching one of the main activities (DOC, BF, STAT).

The following figure 7 shows the injection of the crash using throw error in react-native for the Board Button.



```
Error: This is Injected Crash                    ×

App
C:/React/ExpoCrudBoard/App.tsx:44

    }, [theme.colors, theme.dark]);

    if (!isDidMount) {
>     throw new Error('This is Injected Crash');
  ^} else {
      return (
        <SafeAreaProvider>

View compiled

▶ 19 stack frames were collapsed.

registerRootComponent
C:/src/launch/registerRootComponent.web.tsx:14

    const RootComponent: React.FC<P> = props =>
<App {...props} />;
    AppRegistry.registerComponent('main', () =>
RootComponent);
    const rootTag =
document.getElementById('root') ??
document.getElementById('main');
>   AppRegistry.runApplication('main', { rootTag
});
  }

View compiled

Module.../../../../../../../../../../React/ExpoCrudBo
ard/node_modules/expo/AppEntry.js
C:/React/ExpoCrudBoard/node_modules/expo/AppEntry.js:5

__webpack_require__
C:/React/ExpoCrudBoard/webpack/bootstrap:789
```

Figure 7: Crash Injection in to Expo Crud Board Application

5.2.2 GUI Errors

GUI Changes Error Injection

Our target is detecting GUI changes in (CSS, HTML and Props), so for this application, we injected random colors changes for the background, changing in HTML properties and adding random keys for detecting changes in props, using componentWillUnmount, and componentWillUpdate, the following figure 8,9,10 is a snapshot from the code.

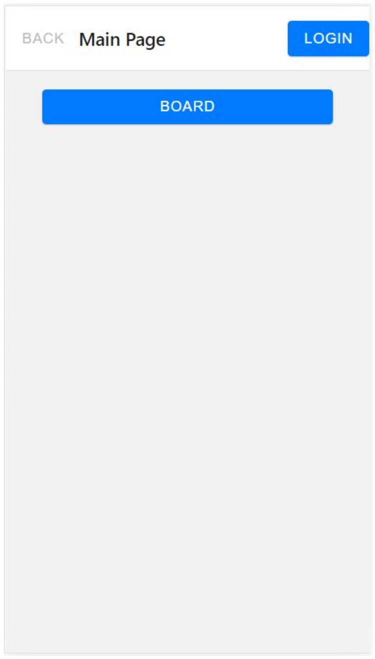


Figure 8: Application Before Random Colors Error Injection

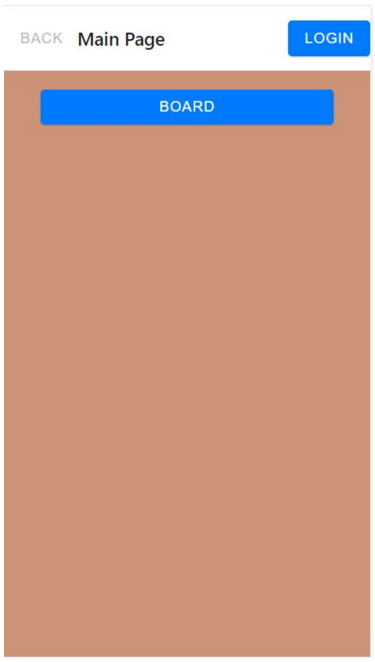


Figure 9: Application After Will Mount

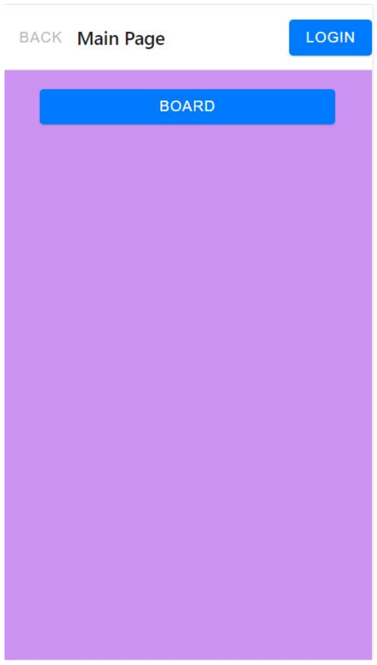


Figure 10: Application After Will Mount

GUI Wrong Component Positions Error Injection

Our target is GUI wrong position detecting after changing in one of the DOC, STAT, or BF events. In this application, we apply some object position changes using componentWillmount and componentWillUpdate, in this application we changed the Login component to be replaced with the Back component and vice versa, as shown in figure 11.

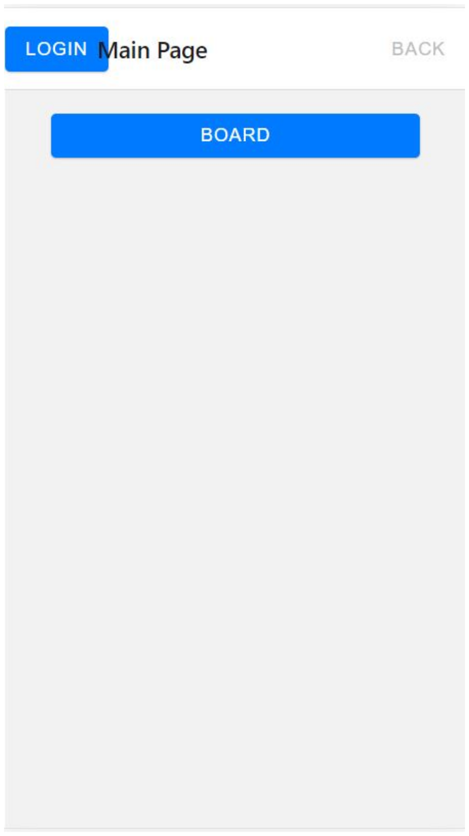


Figure 11: GUI Wrong Component Positions Error Injection After Background Foreground Event

GUI Component Appearance/Disappearance Error Injection

Our target is GUI disappear one of the objects detecting after changing in one of the DOC, STAT, or BF events. In this application, we apply to remove and random adding of objects using componentWillmount and componentWillUpdate, in this application, we remove the Login component randomly after the events, as shown in figure 12.

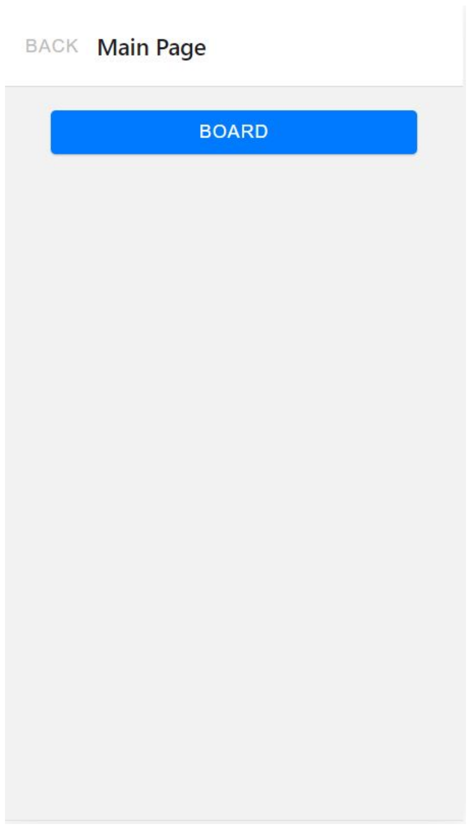


Figure 12: GUI Component Disappearance Error Injection After Background Foreground Event

5.3 Evaluation Execution

We executed the framework in Expo Crud Board without any bugs injection to ensure that the application does not have bugs from executing DOC, STAT and BF events. For the evaluation process, we injected the bugs one by one in each round. For each bug, we included two phases of testing. In the **first phase**, we executed each event (DOC, BF, STAT) for one time; the main goal from this phase is to make sure that each event could detect the bugs with the same efficiency.

The **second phase** included the full cycle of framework execution, which starts with random triggering of the events. For this phase, the main goal is to detect the same bug in each full cycle test round and finalise with results. In addition, we executed our framework in Expo Crud Board application for 10 min for each bug that we injected. For each round, the framework waits for 1 min; this has improved the results we detected in the first round of the evaluation process. Besides, it helped the framework detect and reduce application crashes for unexpected reasons other than lifecycle issues. We saw that the 1 min interval is common used for many frameworks [36] [37].

Ex :

- Inject the GUI changes error ”we already explained this error details in the previous section”; for each event (ex: BF event), the background of the GUI will be changed.
- Add the configuration of the full cycle testing details in the configuration file (in our case add 10 min full-cycle testing)
- The framework lunching BF or DOC event in each min of the configured time.
- In each event our framework will detect the changing error because the background color changed (we inject this error to change the background color after each will mount activity) in each event.
- The framework output is JSON file of the difference between the original GUI and the changed GUI.

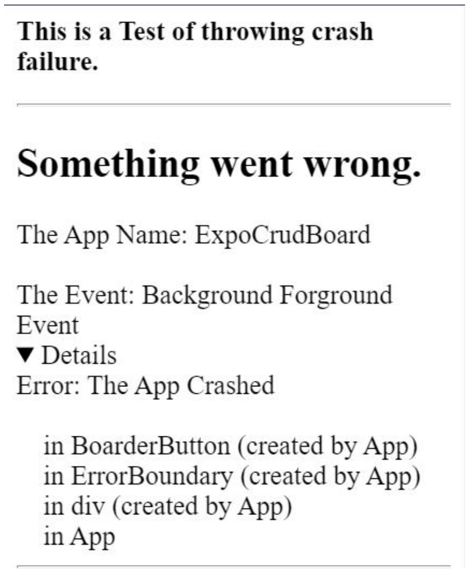
6 Results and Discussion

In this chapter, we will discuss the results of the evaluation that we did using Expo Crud Board application, then we will discuss the results according to the research questions.

6.1 Results

6.1.1 Crash Failure

After Injecting the Crash Failure for the first phase, the BF and DOC events trigger the key loops. The framework output from the crash detecting contains The mobile application name: Expo Crud Board, The sequence of event lifecycle that causes crash which is Background Foreground event, The name of detected failure with the activity name: crashed, also the unhandled exception and the crash stack trace: from component Boarder Buttony, Error Boundary, div, App; as shown in the following figure 13.



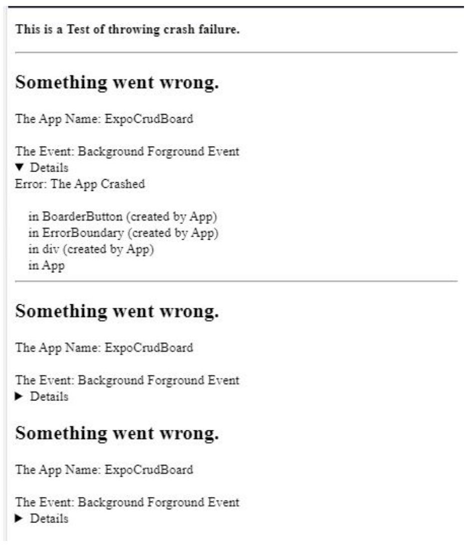
This is a Test of throwing crash failure.

Something went wrong.

```
The App Name: ExpoCrudBoard

The Event: Background Forground
Event
▼ Details
Error: The App Crashed

    in BoarderButton (created by App)
    in ErrorBoundary (created by App)
    in div (created by App)
    in App
```

Figure 13: Crash Report

For the second phase, the full cycle execution, we got 6 out of 10 expected crashes. Then, the framework crashed according to expo limitation of handling multiple crashing. This will be discussed in section 6.2.

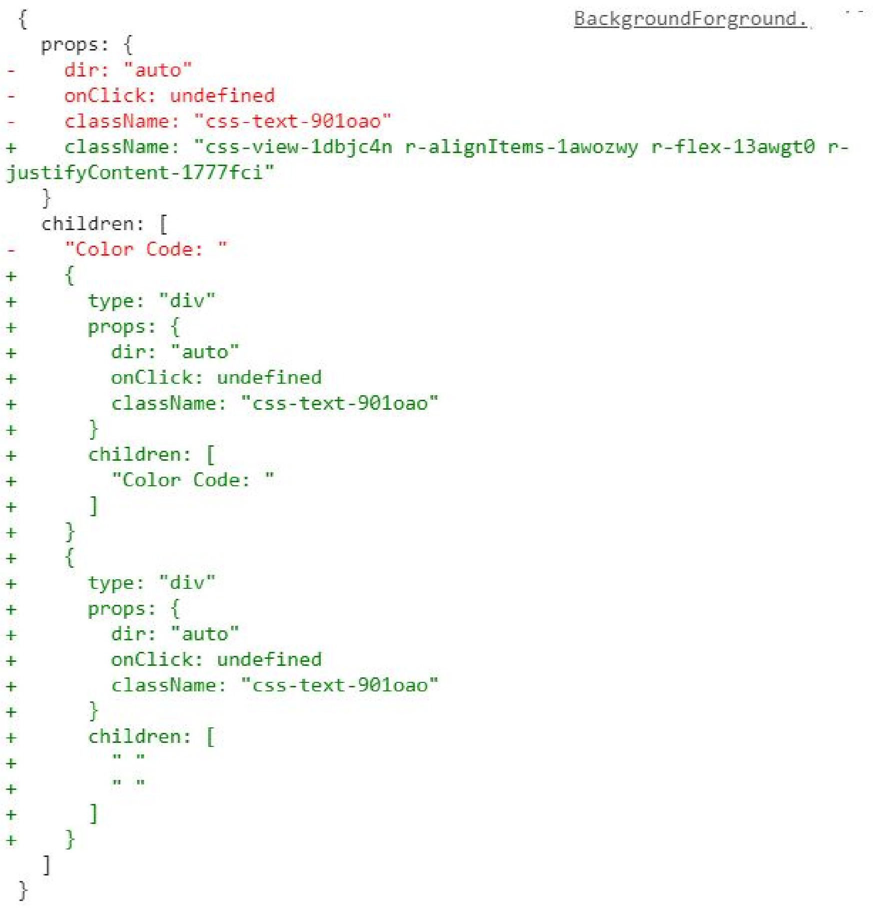


```
This is a Test of throwing crash failure.

Something went wrong.

The App Name: ExpoCrudBoard

The Event: Background Forground Event
▼ Details
Error: The App Crashed

   in BoarderButton (created by App)
   in ErrorBoundary (created by App)
   in div (created by App)
   in App

Something went wrong.

The App Name: ExpoCrudBoard

The Event: Background Forground Event
▶ Details

Something went wrong.

The App Name: ExpoCrudBoard

The Event: Background Forground Event
▶ Details
```

Figure 14: Full Cycle Crash Report

6.1.2 GUI Errors

After Injecting the GUI error for the first phase, which uses BF and DOC events to trigger the key loops, we got the difference in Oracles results with details of each component that has been affected, as shown in the following figure 15,



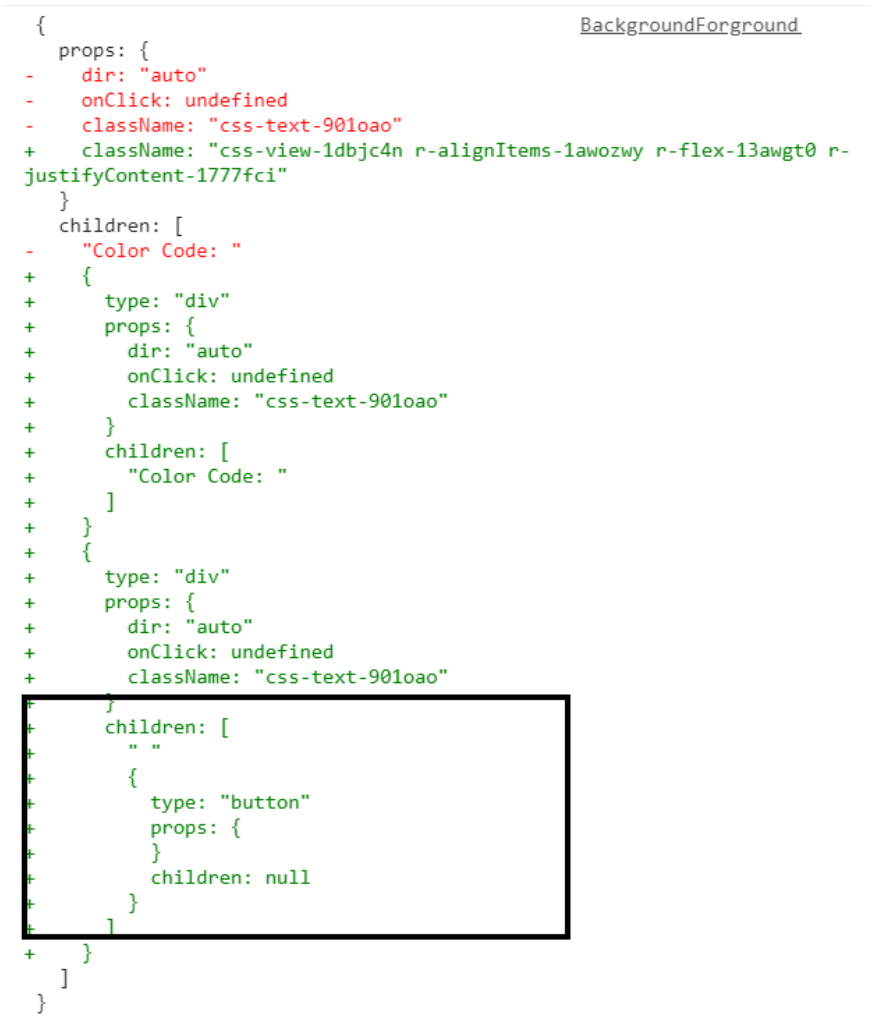
```
 {                                              BackgroundForground.
   props: {
-    dir: "auto"
-    onClick: undefined
-    className: "css-text-901oao"
+    className: "css-view-1dbjc4n r-alignItems-1awozwy r-flex-13awgt0 r-
justifyContent-1777fci"
   }
   children: [
-    "Color Code: "
+    {
+      type: "div"
+      props: {
+        dir: "auto"
+        onClick: undefined
+        className: "css-text-901oao"
+      }
+      children: [
+        "Color Code: "
+      ]
+    }
+    {
+      type: "div"
+      props: {
+        dir: "auto"
+        onClick: undefined
+        className: "css-text-901oao"
+      }
+      children: [
+        " "
+        " "
+      ]
+    }
   ]
 }
```

Figure 15: Changing in GUI Report

This result was detected after applying the bug shown in figures 8,9,10. The main result defines the changes in the GUI before and after applying the BF event. As shown in the figure, the deleted properties from the component are in the red lines, and the added properties are in the green lines.

For the second phase, the full cycle execution, we got 10 (the same bug was detected in each cycle) out of expected 10 changes. These results were from each error we inject from the GUI errors mentioned in the previous section.

We injected two GUI errors (color changing and remove/add components) as an extra phase. The following figure shows all details of the differences.

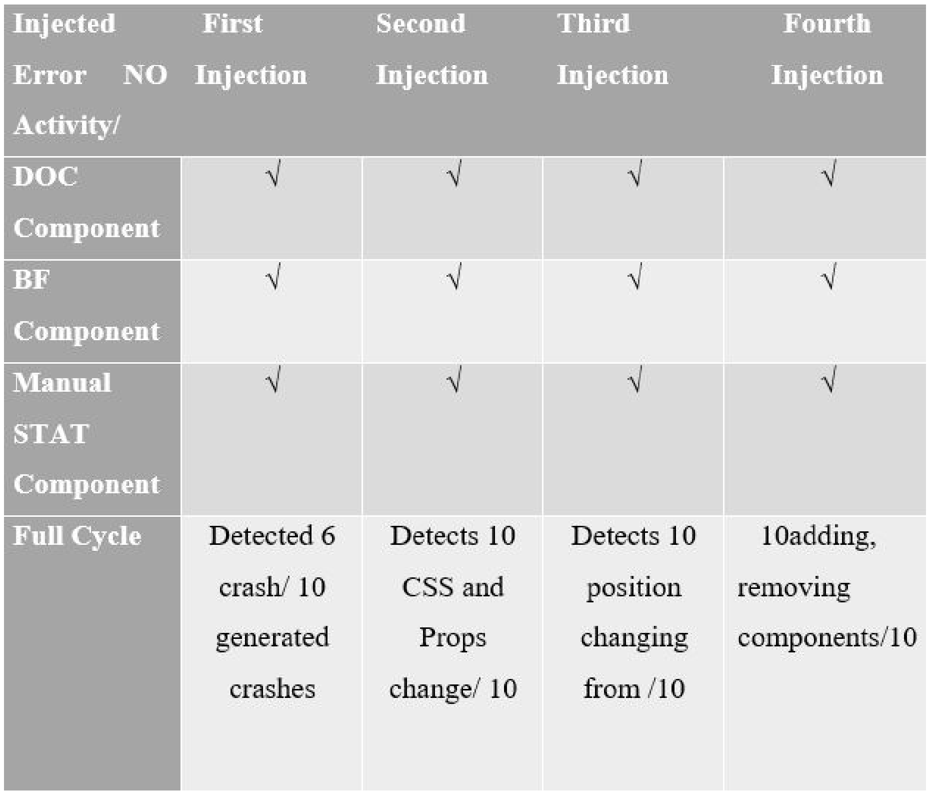


BackgroundForground

```
 {
   props: {
-    dir: "auto"
-    onClick: undefined
-    className: "css-text-901oao"
+    className: "css-view-1dbjc4n r-alignItems-1awozwy r-flex-13awgt0 r-
justifyContent-1777fci"
   }
   children: [
-    "Color Code: "
+    {
+      type: "div"
+      props: {
+        dir: "auto"
+        onClick: undefined
+        className: "css-text-901oao"
+      }
+      children: [
+        "Color Code: "
+      ]
+    }
+    {
+      type: "div"
+      props: {
+        dir: "auto"
+        onClick: undefined
+        className: "css-text-901oao"
+      }
+      children: [
+        " "
+        {
+          type: "button"
+          props: {
+          }
+          children: null
+        }
+      ]
+    }
   ]
 }
```

Figure 16: Changing in GUI Report for Two Injected Errors

Over All Results



Injected Error NO Activity/	First Injection	Second Injection	Third Injection	Fourth Injection
DOC Component	√	√	√	√
BF Component	√	√	√	√
Manual STAT Component	√	√	√	√
Full Cycle	Detected 6 crash/ 10 generated crashes	Detects 10 CSS and Props change/ 10	Detects 10 position changing from /10	10adding, removing components/10

Figure 17: Evaluation Matrix

6.2 Discussion

In this section our main goal is to answer the research questions that we have in this report:

- **RQ1:** How effective is our framework in detecting crashes in the applications that are based on the react-native activity lifecycle?

  Our framework detected the injected crashes using DOC, BF, and STAT events. For the full cycle execution, our framework handled 60% of the crashes. (note: this number was increased according to the first evaluation because of adding 1 min waiting between each execution of the events). The main reason for detecting 60% of the injected crash is the limitation of using expo SDK with Android OS using MI device used in our evaluation; expo SDK with MI device couldn't handle many crashes in the same application according to the full-stack issue.

- **RQ2:** How effective is our framework in detecting GUI errors of wrong positions, change object properties (props, CSS, HTML),disappear objects that are based on the react-native activity lifecycle?

  Wrong Positions

  Our results about this type of error show that 100% of the changed positions were detected; all these changes were detected as CSS changes. We removed the limitation of the phone type in this application using some timers for 1 min between each activity by specifying the number of triggering events to 10. In addition, we were in need to detect changes in GUI errors when we used each component of DOC, Manual STAT, BF activities.

  Change Object Properties (props, CSS, HTML)

  In this type of error, we can see that 100% of all changes application, we removed the limitation of the phone type in this application using some timers for 1 min between each activity by specifying the number of triggering events to 10. In addition, we need to detect changes in GUI errors when we used each component of DOC, Manual STAT, BF activities.

disappear objects

In this type of error, we can see that 100% (note: we removed added percentage by test the application before starting the framework evaluation) in GUI when we tested DOC, Manual STAT, BF components once.

6.3 Threats to Validity

This section will discuss the threats that could affect our results that we obtained in our study.

6.3.1 Internal Validity

Some failures that we detected using our framework might have been caused by different reasons instead of life cycle sequence events.

These failures could be caused by alternative factors, such as the execution platform or the timing between consecutive events. To mitigate this threat, detected failure was manually reproduced on an actual device to exclude being tied to the testing infrastructure during the validation step. A controlled experiment involving different devices and time intervals between events should be carried out to investigate this aspect further.

6.3.2 External Validity

In our research, we had a small set of proper benchmarks for our framework testing. This small set may affect the generalizability of the evaluation results. We should figure the framework evaluation with a larger set of applications in the future, also with different sets of SDKs. In addition, the author injected the failures, which could bias results.

7 Conclusion and Future Work

In this research, we present an automated lifecycle testing framework for react-native mobile applications. Our framework approach focused on react-native mobile application crashes and GUI error testing. Besides, our framework is the first framework that handles react-native lifecycle testing. In this research, we handle the background of our work, such as react-native as a cross-platform library and react-native lifecycle. Also, we provide a literature review about general mobile applications testing, cross-platform mobile application testing, react-native comparison with the other platforms, and mobile application lifecycle testing.

For future work, we can evaluate our framework using a real set of applications. Therefore, we can improve our results and the implemented code. In addition, we can cover the iOS OS in our framework to help the react-native developer to have full stress testing in Android and iOS OSs.

References

[1] S. O. Dea. (2020). "Number of smartphone subscriptions worldwide from 2016 to 2026," [Online]. Available: `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`. accessed: 11.12.2020.

[2] S. Counter. (2020). "Mobile operating system market share worldwide," [Online]. Available: `https://gs.statcounter.com/os-market-share/mobile/worldwide`. accessed: 24.10.2020.

[3] W. Danielsson, *React native application development: A comparison between native android and react native*, 2016.

[4] N. Kousar, M. Sheraz, A. Malik, A. Sarwar, B. Mohy-ud-din, and A. Shahid, "Software engineering: Challenges and their solution in mobile app development," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 1, pp. 200–203, 2018.

[5] D. Amalfitano, V. Riccio, A. C. Paiva, and A. R. Fasolino, "Why does the orientation change mess up my android application? from gui failures to code faults," *Software Testing, Verification and Reliability*, vol. 28, no. 1, e1654, 2018.

[6] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE international conference on software testing, verification and validation (icst)*, IEEE, 2016, pp. 33–44.

[7] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in android applications," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 864–880, 2016.

[8] S. Zein, N. Salleh, and J. Grundy, "A systematic mapping study of mobile application testing techniques," *Journal of Systems and Software*, vol. 117, pp. 334–356, 2016.

[9] G. Hu, X. Yuan, Y. Tang, and J. Yang, “Efficiently, effectively detecting mobile app bugs with appdoctor,” in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–15.

[10] M. Staats, M. W. Whalen, and M. P. Heimdahl, “Programs, tests, and oracles: The foundations of testing revisited,” in *2011 33rd international conference on software engineering (ICSE)*, IEEE, 2011, pp. 391–400.

[11] V. Riccio, D. Amalfitano, and A. R. Fasolino, “Is this the lifecycle we really want? an automated black-box testing approach for android activities,” in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, 2018, pp. 68–77.

[12] D. Rimawi and S. Zein, “A model based approach for android design patterns detection,” in *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, IEEE, 2019, pp. 1–10.

[13] B. Eisenman, *Learning react native: Building native mobile apps with JavaScript.* ” O’Reilly Media, Inc.”, 2015.

[14] S. Liu. (2020). “Cross-platform mobile frameworks used by developers worldwide 2019 and 2020,” [Online]. Available: `https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/`. accessed: 20.1.2021.

[15] S. Koper. (2021). “A good start in react native,” [Online]. Available: `https://www.netguru.com/codestories/a-good-start-in-react-native`. accessed: 10.1.2021.

[16] A. Paul and A. Nalwaya, “React native for mobile development,” *React Native for Mobile Development. California: Apress, Berkeley, CA. https://doi. org/10.1007/978-1-4842-4454-8*, 2019.

[17] M. Kuitunen, “Cross-platform mobile application development with react native,” B.S. thesis, 2019.

[18] D. Makai, *Expo v/s react native cli. what to choose and what to stay away from.* Jul. 2021. [Online]. Available: `https://medium.com/nerd-for-tech/expo-v-s-react-native-cli-what-to-choose-and-what-to-stay-away-from-85afc81597bc`.

[19] N. Zamin, N. M. Norwawi, N. Arshad, D. Rambli, *et al.*, "Make me speak: A mobile app for children with cerebral palsy," *International Journal of Advanced Trends in Computer Science and Engineering*, 2019.

[20] L. Raymond. (2021). "The react component lifecycle," [Online]. Available: `https://medium.com/@louis.raymond/the-react-component-lifecycle-ccfff518e7eb`. accessed: 25.1.2021.

[21] M. Salohonka, "Automated testing of react native applications," *Lappeenranta Lahti University of Technology LUT*, 2020.

[22] C. Denger and T. Olsson, "Quality assurance in requirements engineering," in *Engineering and managing software requirements*, Springer, 2005, pp. 163–185.

[23] B. Amen, S. Mahmood, and J. Lu, "Mobile application testing matrix and challenges," *Computer Science & Information Technology*, pp. 27–40, 2015.

[24] T. Wetzlmaier, R. Ramler, and W. Putschögl, "A framework for monkey gui testing," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2016, pp. 416–423.

[25] P. Nidagundi and L. Novickis, "New method for mobile application testing using lean canvas to improving the test strategy," in *2017 12th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT)*, IEEE, vol. 1, 2017, pp. 171–174.

[26] K. S. Arif and U. Ali, "Mobile application testing tools and their challenges: A comparative study," in *2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, IEEE, 2019, pp. 1–6.

[27] R. R. Nimbalkar, "Mobile application testing and challenges," *International Journal of Science and Research*, vol. 2, no. 7, pp. 56–58, 2013.

[28] K. Bordi, "Overview of test automation solutions for native cross-platform mobile applications," *Computer Science & Information Technology*, 2018.

[29] A. A. Menegassi and A. T. Endo, "Automated tests for cross-platform mobile apps in multiple configurations," *IET Software*, vol. 14, no. 1, pp. 27–38, 2019.

[30] M. Ciman and O. Gaggi, "An empirical analysis of energy consumption of cross-platform frameworks for mobile development," *Pervasive and Mobile Computing*, vol. 39, pp. 214–230, 2017.

[31] N. Hansson and T. Vidhall, *Effects on performance and usability for cross-platform application development using react native*, 2016.

[32] M. Furuskog and S. Wemyss, *Cross-platform development of smartphone applications: An evaluation of react native*, 2016.

[33] O. Riganelli, S. P. Mottadelli, C. Rota, D. Micucci, and L. Mariani, "Data loss detector: Automatically revealing data loss bugs in android apps," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 141–152.

[34] N. Hoshieah, S. Zein, N. Salleh, and J. Grundy, "A static analysis of android source code for lifecycle development usage patterns," *Journal of Computer Science*, vol. 15, no. 1, pp. 92–107, 2019.

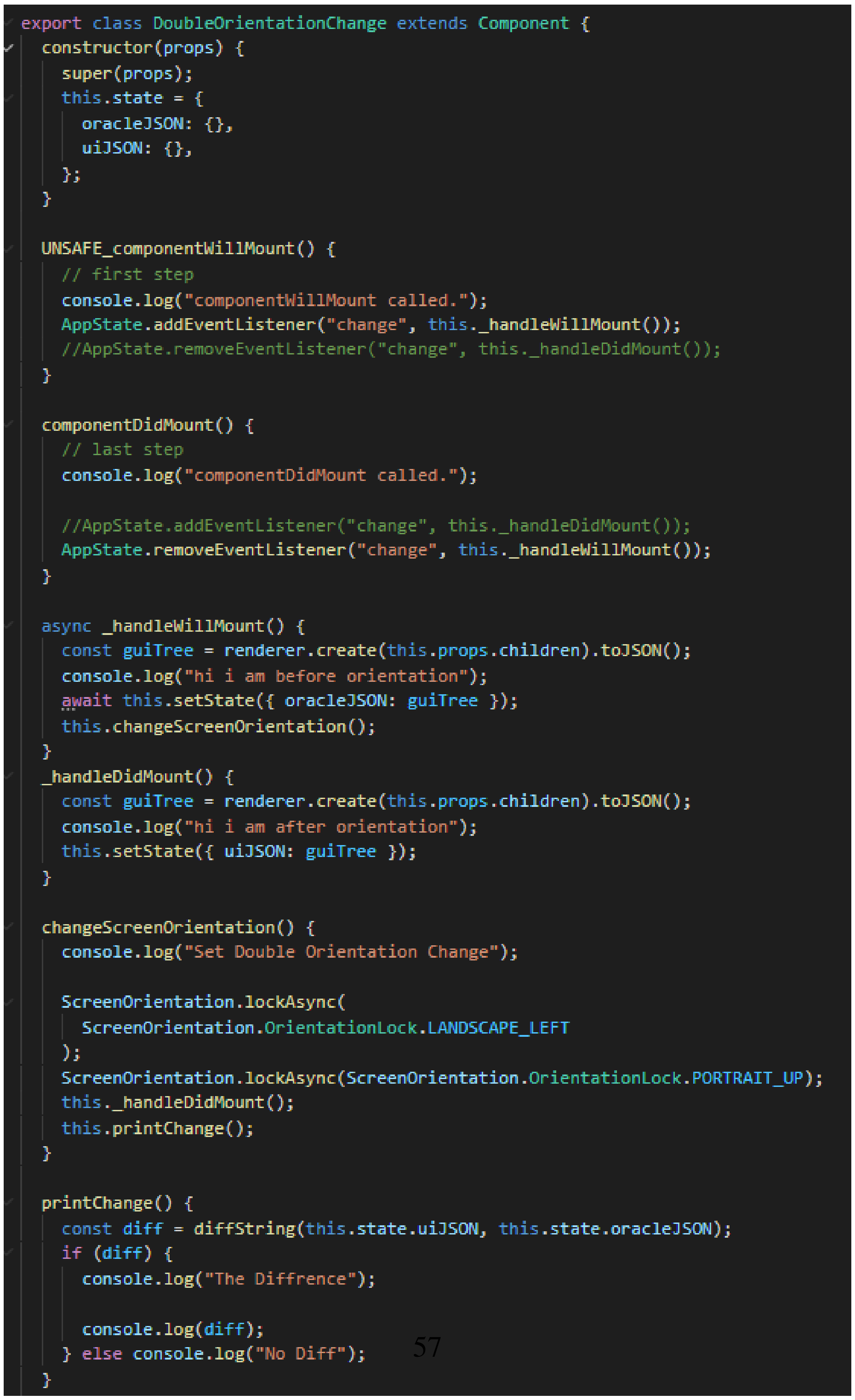
[35] S. Zein, N. Salleh, and J. Grundy, "Static analysis of android apps for lifecycle conformance," in *2017 8th International Conference on Information Technology (ICIT)*, IEEE, 2017, pp. 102–109.

[36] M. W. Levin, K. M. Kockelman, S. D. Boyles, and T. Li, "A general framework for modeling shared autonomous vehicles with dynamic network-loading and dynamic ride-sharing application," *Computers, Environment and Urban Systems*, vol. 64, pp. 373–383, 2017.

[37] A. E. Xhafa and O. K. Tonguz, “Dynamic priority queueing of handover calls in wireless networks: An analytical framework,” *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 5, pp. 904–916, 2004.

A Appendix

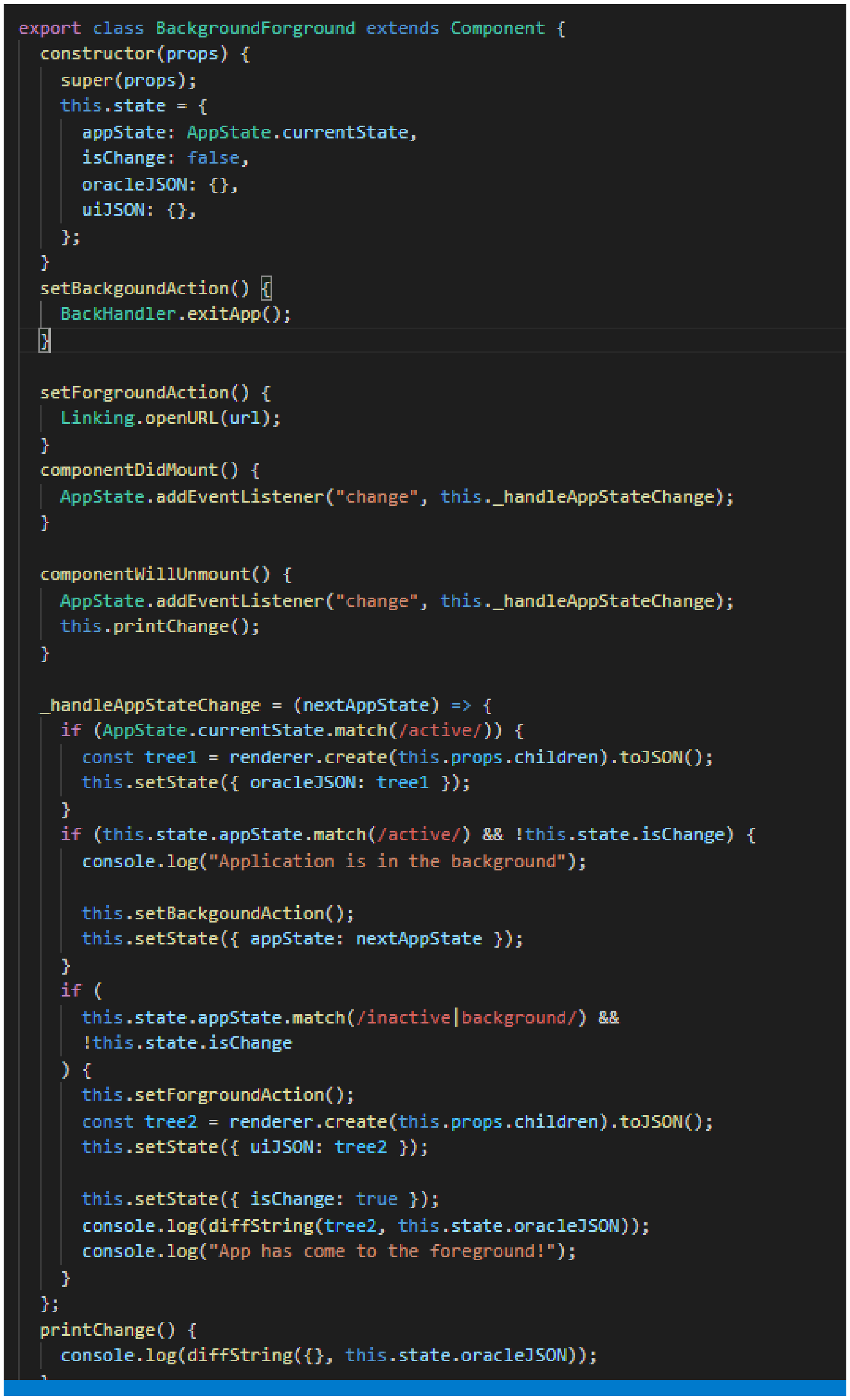
A.1 DOC Component logical code



```
export class DoubleOrientationChange extends Component {
  constructor(props) {
    super(props);
    this.state = {
      oracleJSON: {},
      uiJSON: {},
    };
  }

  UNSAFE_componentWillMount() {
    // first step
    console.log("componentWillMount called.");
    AppState.addEventListener("change", this._handleWillMount());
    //AppState.removeEventListener("change", this._handleDidMount());
  }

  componentDidMount() {
    // last step
    console.log("componentDidMount called.");

    //AppState.addEventListener("change", this._handleDidMount());
    AppState.removeEventListener("change", this._handleWillMount());
  }

  async _handleWillMount() {
    const guiTree = renderer.create(this.props.children).toJSON();
    console.log("hi i am before orientation");
    await this.setState({ oracleJSON: guiTree });
    this.changeScreenOrientation();
  }
  _handleDidMount() {
    const guiTree = renderer.create(this.props.children).toJSON();
    console.log("hi i am after orientation");
    this.setState({ uiJSON: guiTree });
  }

  changeScreenOrientation() {
    console.log("Set Double Orientation Change");

    ScreenOrientation.lockAsync(
      ScreenOrientation.OrientationLock.LANDSCAPE_LEFT
    );
    ScreenOrientation.lockAsync(ScreenOrientation.OrientationLock.PORTRAIT_UP);
    this._handleDidMount();
    this.printChange();
  }

  printChange() {
    const diff = diffString(this.state.uiJSON, this.state.oracleJSON);
    if (diff) {
      console.log("The Diffrence");

      console.log(diff);
    } else console.log("No Diff");
  }
```

Figure 18: DOC Component logical code

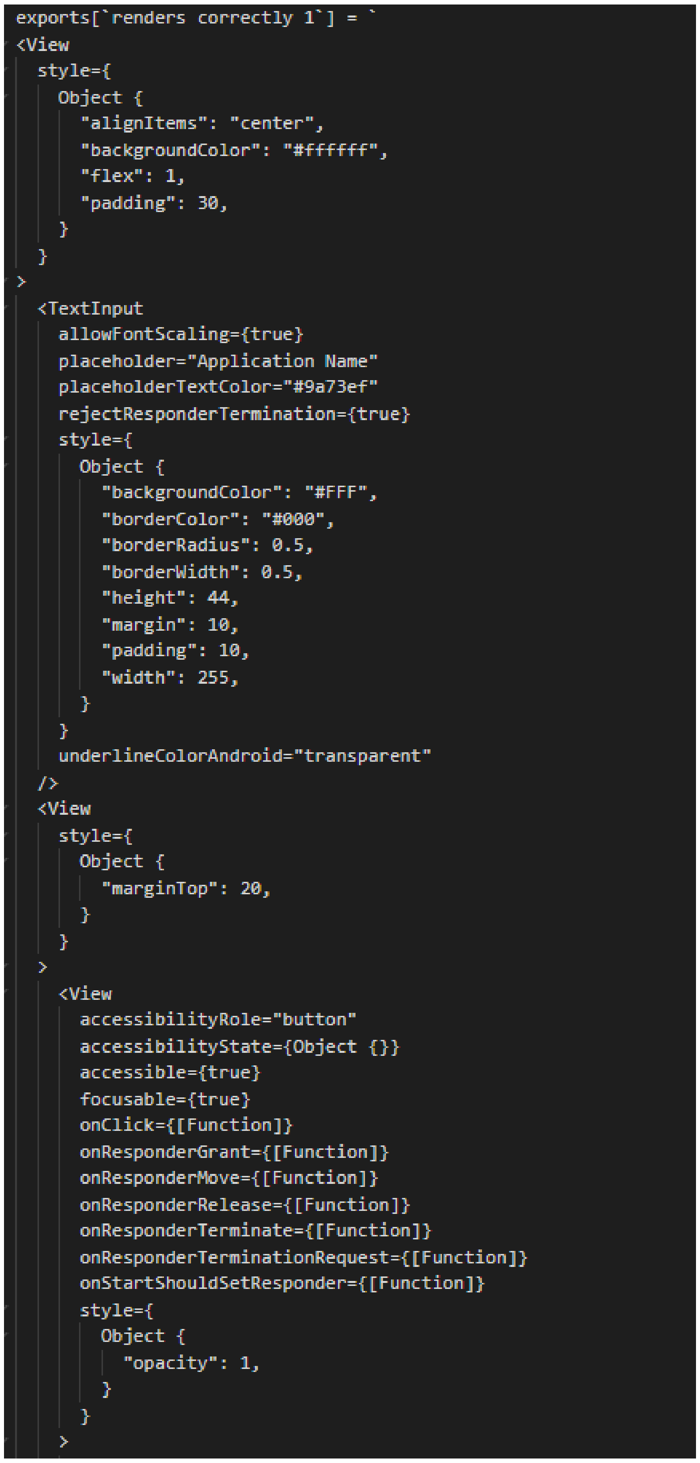
A.2 Background Foreground Component



```
export class BackgroundForground extends Component {
  constructor(props) {
    super(props);
    this.state = {
      appState: AppState.currentState,
      isChange: false,
      oracleJSON: {},
      uiJSON: {},
    };
  }
  setBackgoundAction() {
    BackHandler.exitApp();
  }

  setForgroundAction() {
    Linking.openURL(url);
  }
  componentDidMount() {
    AppState.addEventListener("change", this._handleAppStateChange);
  }

  componentWillUnmount() {
    AppState.addEventListener("change", this._handleAppStateChange);
    this.printChange();
  }

  _handleAppStateChange = (nextAppState) => {
    if (AppState.currentState.match(/active/)) {
      const tree1 = renderer.create(this.props.children).toJSON();
      this.setState({ oracleJSON: tree1 });
    }
    if (this.state.appState.match(/active/) && !this.state.isChange) {
      console.log("Application is in the background");

      this.setBackgoundAction();
      this.setState({ appState: nextAppState });
    }
    if (
      this.state.appState.match(/inactive|background/) &&
      !this.state.isChange
    ) {
      this.setForgroundAction();
      const tree2 = renderer.create(this.props.children).toJSON();
      this.setState({ uiJSON: tree2 });

      this.setState({ isChange: true });
      console.log(diffString(tree2, this.state.oracleJSON));
      console.log("App has come to the foreground!");
    }
  };
  printChange() {
    console.log(diffString({}, this.state.oracleJSON));
```

Figure 19: Background Foreground Component

A.3 Oracles As a JSON



```
exports[`renders correctly 1`] = `
<View
  style={
    Object {
      "alignItems": "center",
      "backgroundColor": "#ffffff",
      "flex": 1,
      "padding": 30,
    }
  }
>
  <TextInput
    allowFontScaling={true}
    placeholder="Application Name"
    placeholderTextColor="#9a73ef"
    rejectResponderTermination={true}
    style={
      Object {
        "backgroundColor": "#FFF",
        "borderColor": "#000",
        "borderRadius": 0.5,
        "borderWidth": 0.5,
        "height": 44,
        "margin": 10,
        "padding": 10,
        "width": 255,
      }
    }
    underlineColorAndroid="transparent"
  />
  <View
    style={
      Object {
        "marginTop": 20,
      }
    }
  >
    <View
      accessibilityRole="button"
      accessibilityState={Object {}}
      accessible={true}
      focusable={true}
      onClick={[Function]}
      onResponderGrant={[Function]}
      onResponderMove={[Function]}
      onResponderRelease={[Function]}
      onResponderTerminate={[Function]}
      onResponderTerminationRequest={[Function]}
      onStartShouldSetResponder={[Function]}
      style={
        Object {
          "opacity": 1,
        }
      }
    >
```

Figure 20: Oracles As a JSON