Journal of Computer Science

Original Research Paper

# Testing React Native Mobile Apps: Pruning GUI Model Approach

**[1]Rand Marwan Khalil Ibrahim and [2]Samer Zein**

*[1]Department of Information Technology, Birzeit University, Birzeit, Palestinian Authority*
*[2]Department of Computer Science, Birzeit University, Birzeit, Palestinian Authority*

*Article history*
Received: 09-09-2023
Revised: 20-11-2023
Accepted: 01-01-2024

Corresponding Author:
Rand Marwan Khalil Ibrahim
Department of Information Technology, Birzeit University, Birzeit, Palestinian Authority
Email: randibrahim94@gmail.com

**Abstract:** The mobile app market is still in continual growth. People are migrating to smartphone mobile devices to accomplish their daily activities while working, playing, and communicating with others. From the developers' perspective, there exists a wide variety of platforms, technologies, and architecture choices for developing and testing mobile apps. However, because of the constant changes in software applications and the great technological development, developers are supposed to speed up the development process to satisfy the customer's needs and provide robust applications within a short period of time. Cross-platform mobile app development technology, such as react Native, aims to overcome these difficulties, where instead of building separate applications for each platform, a single code base that can be run on multiple platforms is developed, which accelerates the development process. Model-based testing is one of the techniques that are used to test cross-platform applications and identify and find defects and bugs. This study proposes a React Native Abstract Syntax Tree pruning (RN-AST pruning) framework, which aims to facilitate the mobile app testing process by pruning the original GUI model of the application and reducing the number of test cases by keeping only the test cases that cover the impacted regions from internal code changes. The pruning process to keep the GUI elements is applied to the abstract syntax tree, which is the result of doing the static analysis on the last two versions of the source code. After that the two pruned AST will be compared to keep only the affected and updated GUI elements. The affected files will be listed as paths to prevent any other file from being tested, consequently reducing the number of test cases. According to our knowledge, no comprehensive work was dedicated to use the static analysis approach in keeping only the impacted GUI elements by the internal code changes in cross-platform software, thus reducing the run test cases and increasing productivity by accelerating the development life cycle. Preliminary experimentation was done on our framework with the help of six developers and test engineers in cross-platform development. The experiment was carried out in a systematic process with clear steps on a proof of concept mobile application. Results show that the RN-AST Pruning framework is useful and provides test engineers with affected files and paths that need to be tested, thus reducing the test cases and minimizing the testing time and effort. Moreover, it identifies exactly the changes that occurred in each file and categorizes them into updates, placements, and deletions based on the differences between the original version and the updated version of the source code. The authors confirm that this study is original and its contents are unpublished. Moreover, no specific grant from any funding agency was received.

**Keywords:** React Native, GUI Model, Pruning, Testing Process

## Introduction

Nowadays, with the heavy reliance on technology and due to technological development, mobile application development is evolving rapidly and moving quickly toward being mainstream. Mobile devices and smartphones are becoming an integral part of our modern life. Almost 60% of the population is accessing the internet using their mobile devices, which increases the need to support the ongoing development in this area. Theoretically, this seems to be easy, however, technically this is complex because of the rapid development nature and imposed limitations for mobile devices (Hartmann *et al*., 2011), especially since user expectations about mobile applications are remarkably high (Raj and Tolety, 2012).

In general, the diversity of mobile platforms makes the mobile development process quite complex and expensive, particularly with the need to build the application for each mobile operating system. This raises the necessity to have a cross-platform development to contribute to solving this diversity problem.

Such applications give developers the ability to launch software simultaneously on various platforms, making the development process faster than before because as a developer you need to deploy only a single script to run against different platforms. Moreover, it offers the opportunity to reach a wide range of audiences. Furthermore, using cross-platform saves money and time, where the time to market will be reduced, which increases the application revenues. Despite the huge advantages, using cross-platform technologies still has downsides and limitations. The largest risk is the maturity of this technology (Eisenman, 2015), as cross-platform development is still relatively young. Moreover, some features of iOS and Android still aren't supported and different practices are still under process. Despite the great adoption of cross-platform development and the rapid evolvement in this field, particularly the react Native framework, there is still a lack in the existence of frameworks that assist in testing the graphical user interfaces in mobile applications and provide the test engineers with a subset of test cases to check and run instead of testing the whole test cases. Accordingly, there is a need to exploit model-based testing in building a framework that can assist the test engineers in testing the graphical user interfaces, which have become a nearly ubiquitous means of interacting with software systems (Memon, 2002). The framework is based on the idea of pruning the entire model to guide and help the test engineers in reaching the modified GUI parts of the application impacted by the internal changes on the source code.

RN-AST Pruning framework aims to detect the GUI elements, find the GUI changes between the last two versions of the source code, and build the list of paths to be tested. Therefore, reduce the number of test cases that ensure that the application satisfies the needs and requirements. Results showed the effectiveness of the framework in detecting the changes between the source codes and classifying these changes to facilitate the testing process.

The research addressed four questions. First, how to detect the GUI elements and prune the GUI model? Second, how to calculate and classify the code differences and changes between the last two versions of the application? Third, how to build the list of paths that contain the changes file? Fourth, how effective is the framework in detecting changes and results that satisfy the test engineers?

To the best of our knowledge, no studies cover the static analysis and GUI model pruning to facilitate the testing. However, there are some existing searches on model-based testing and GUI model pruning that are relevant to our present study.

In their study Salihu *et al*. (2017) proposed a hybrid technique to support the reverse engineering of the GUI model of the mobile application. The basic idea of their technique was to use both static and dynamic analysis; the GUI Information was extracted using the static analysis for the byte code of the application and then a dynamic crawling was done to reverse engineering the GUI model of the application. Their static analyzer took the application APK as an input and started the analysis process to end up with a Window Transition Graph (WTG), which is made of nodes (GUI widgets) and edges(Events). This graph then entered the dynamic crawler to extract the GUI widgets and their related events to produce the GUI state model as an output. They aimed to clarify the GUI behavior using an effective and high-quality model. They made a prototype called AMOGA for their study that used the hybrid approach to generate a model to describe the behavior of a mobile application. This model can be used to generate test cases to test that application.

Another study that discussed the importance of model-based testing for mobile applications is the orbit tool (Yang *et al*., 2023). This study is an automated GUI-model generator for mobile applications. The proposed work used static analysis on the mobile application source code in order to extract the different events and actions supported by each GUI widget. Then, these events were exercised live using a dynamic crawler to identify the GUI behavior of the application. Identifying the different actions and events in the static analysis involved three basic steps: (1) Identify where the action is registered or instantiated, (2) Locate the GUI component on which the event is fired, (3) Determine the component identifier to help the dynamic analysis in recognizing the component and firing the action.

Tao and Gao (2016), discussed the rapid evolution of mobile and wireless technology, which brought new challenges and issues in the automatic mobile testing process. One of the biggest issues is the lack of mobile test scripting techniques and tools that can deal with the diversity of mobile test environments and devices. Therefore, they introduced a new tool based on GUI ripping to facilitate the validation of numerous mobile applications. They provided a large-scale automation solution by incorporating different open-source technologies like Appium and Selenium. Their approach can increase the test coverage by allowing the parallel execution of test scripts on multiple mobile devices running on different platforms.

Sebastian Bauersfeld ensures the importance of having a robust and high-quality GUI due to the huge evolution in tablets, and smartphones and the heavy reliance on them to achieve our daily life activities. Testing these GUI applications is still a challenge, whereas manual testing is an expensive, limited, and time-consuming process especially when doing regression tests. Therefore, Sebastian proposed a new regression testing tool for GUI applications which is called GUIDiff (Bauersfeld, 2013). The basic idea behind GUIDiff is to run the two versions of the application in parallel and report the differences between the GUI states to the testers. Therefore, the GUI state information should be captured in widget trees. After that, the two versions of the same application are run side by side to notice the differences between the states in the widget tress. Doing so will compare the properties of the same controls against each other.

## Materials and Methods

Our approach is mainly based on generating a pruned model to facilitate the testing process of React Native applications by reducing the number of test cases required to ensure that the application satisfies the customer's needs. Figure 1 illustrates the proposed structure for the RN-AST pruning framework. As seen, the framework is composed of three parties communicating with each other. The test engineer uses the front-end interface to connect to the back-end side that sends data to the Mongo database, which is considered the third party.

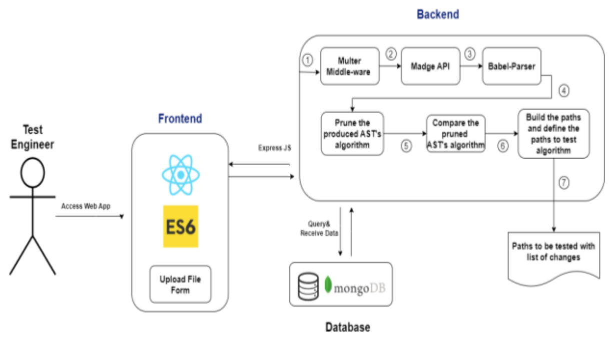
Backend
Multer Middle-ware
Madge API
Babel-Parser
Prune the produced AST's algorithm
Compare the pruned AST's algorithm
Build the paths and define the paths to test algorithm
Test Engineer
Frontend
Express JS
Access Web App
ES6
Upload File Form
Query& Receive Data
mongoDB
Database
Paths to be tested with list of changes

**Fig. 1:** Structure diagram of the framework

The back-end side represents and clarifies briefly the different steps for our framework. First, Multer, which is a node.js middle ware used to handle the process of uploading the source code file from the test engineer side. Second, Madge API generates the visual graph of the dependencies of the uploaded source code. This API has different features that facilitate the process of determining the different dependencies between the different modules in the application, finding circular dependencies, and providing useful information. Then, the babel-parser was chosen to parse the uploaded ECMAScript source code. This parser is a JavaScript parser used in the Babel compiler and it is heavily based on the Acorn JS parser. The parser produced the Abstract Syntax Tree (AST), which is a tree representation of the abstract syntactic structure of the uploaded source code. After producing the AST, it is time to prune this tree and keep only the GUI elements and this is the responsibility of the pruning algorithm. This algorithm will be applied to the produced AST of the uploaded source, which is the original source code and the updated version of that code. After pruning the two ASTs, the comparison algorithm starts its job by comparing the pruned ASTs to find the GUI elements that differentiate between the two versions of the source code. This algorithm classifies the change between the two versions into three classifications:

- Update: That means the same elements but different properties or values
- Placements: That identifies the newly inserted elements in the updated version not in the original version
- Deletions: That tags the deleted elements which are found in the original source code, not in the updated version

Finally, as indicated in step number 7, the paths that contain the changed files are generated and returned to the test engineer who will use them to reduce the testing time and facilitate the testing process.

The below sub-sections describe and explain the different algorithms and steps of our approach in more detail.

### *A Model for GUI*

The test engineer is asked to upload the source code file of the application, which is the starting point for React Native applications. The Multer middleware adds the file object to the request object. Then it comes the time to build and model the structure of the application by building the component diagram and the dependency graph.

In the RN-AST pruning framework, the Madge API was used to generate the visual graph of our application dependencies. Madge API takes the uploaded file, produces the dependency graph based on the imports in the file, and then sends the content of the produced graph as base64 encoding representation to the client side, which on his part, shows the image of the dependency graph to the test engineer.

## *Parse the Code, Detect the JSX Elements, and Prune the AST*

In general, code parsing is the process of breaking up the code sentences or groups of words into separate components based on the set of rules and grammars for each programming language (Utkin *et al*., 2022), where the output of parsing the source code is represented in a tree-like object usually called the abstract syntax tree.

In our study, Babel JavaScript Parser was used to produce the AST, do the source code transformations, and extract the dependencies of the source code as an object.

Both source codes of the original code and the updated code are being parsed using the Babel parser to produce the original AST and the updated AST. These ASTs are pruned to keep only the exported JSX elements for each source code. These JSX elements are the elements that are shown on the user interface of the application. Doing so helps in finding the GUI changes between the two versions of the code. The pseudo-code for the proposed algorithm for pruning the AST of the source code answers the first question and it can be described as shown below.

**Algorithm 1:** L AST pruning and detection algorithm

**Input:** The AST of the source code as array (produced by babel parser)
**Output:** Pruned AST only with JSX elements shown on the screen returned as array
**Steps:**

1) Get the Abstract Syntax Tree of the uploaded source code from the Babel parser
2) Use babel-traverser to traverse the AST nodes and especially the **ExportDefaultDeclaration** node to check the type of default export
3) Get the first rendered GUI element on the screen based on the used export default pattern:

    a) When the export function is the default export after the function declaration, then the algorithm gets the name of the exported function and traverses all the **Function-Decalaration** nodes until the name of the function in the node matches the name of the exported function. Then the first rendered node is the first JSX element stored in the **ReturnStatement** node in the body of the function declaration node
    b) When exporting a variable as default export after the variable declaration, the steps of getting the first rendered JSX element are as above steps. However, instead of traversing the FunctionDeclaration nodes, the algorithm traverses the VariableDeclaration nodes, checks if the variable name matches the export, and then gets the first JSX element from the **ReturnStatement** node in the body of the matched VariableDeclaration node
    c) When exporting a class as default export after the class declaration, the steps of getting the first rendered JSX element are as above steps. However, instead of traversing the FunctionDeclaration or VariableDeclaration nodes, the algorithm traverses the **ClassDeclaration** nodes, gets the different class methods, and then gets the first JSX element from the **Return Statement** node from the **render** method
    d) When exporting **regular syntax** function as default export, the first JSX element would be from the **Return-Statement** node from the body of the **FunctionDeclaration** node
    e) However, when exporting the **arrow syntax** function as default export, the algorithm gets the **ReturnStatement** node as the first JSX element from the body of the arrow function

## *Comparing the JSX Elements Tree of the Original and Updated Source Code*

After producing the pruned ASTs of the original source code and the updated version of the source code, it's time to answer the second question and compare these two ASTs to find the set of differences and this is the aim of this phase.

Below is the pseudo-code for the proposed algorithm for comparing the two pruned ASTs. It is inspired by the reconciliation algorithm in React that deals with figuring out how to update the UI of the application effectively and without any delays.

**Algorithm 2:** Diffing Algorithm

**Input:** The pruned AST of the original source code (old AST) and the pruned AST of the updated source code (new AST)
**Output:** The deletion elements array, the placement elements array, and the updated elements array.
**Steps :**

```
 1: procedure COMPARETWOASTS (oldAST, newAST, update, placement, deletion)
 2:     placement ← ∅; deletion ← ∅; update ← ∅
 3:     if oldAST = ∅ && newAST = ∅ then
 4:         return empty
 5: else if oldAST = ∅ then
 6:         placement ← newAST
 7: else if newAST = ∅ then
 8:         deletion ← of oldAST
 9: else
10:        deletedIds = oldIds. filter (x = > !newIds) includes(x));
```

```
11:    intersectionIds = newIds. filter (x => oldIds.
       includes(x));
12:    placementIds = newIds.filter(x =>!oldIds.
       includes(x))
13:    deletion← deletedId′sobject
14:    placement ← newId′sobject intId ∈
       ⟩nter sectionIds
15:      oldObj = oldAST.find(x => x.props.id === i);
16:      newObj = newAST. find(x     => x.props.
         id === i);
17:      sameType = newObj.name == oldObj.name
18:      if    sameType    &&
         (!lodash.isEqual(oldObj.props,newObj.p
         newObj.value) then
19:          update.push(oldObj);
20:      end if
21:      if newObj && !sameType then
22:      placement. push(newObj)
23:      end if
24:      if oldObj && !sameType then
25:      deletion.push(oldObj)
26:      end if
27:      repeat for children
```

Note that checking the id attributes enhances the comparison algorithm performance and provides the test engineers with clear results. Therefore, the algorithm generates an id attribute for elements that have no id by hashing the type of the element, and its index so ends with an element with a unique id among its siblings.

### *Building the Different Paths of the Dependency Graph*

Technically, a dependency graph is a collection of entities called nodes; in our case, these nodes represent the different files in our application. Generally, nodes are connected by edges that manage the relationship between them. Going through these nodes produces the different paths of the dependency tree starting from the first node, which is the root, and ending with leaves, which are the nodes with no dependencies. In the RN-AST Pruning framework, the idea of getting the different paths of the dependency graph is based on traversing the graph using the Depth-First-Search *(*DFS*)* technique. The DFS algorithm starts at the root node and explores as far as possible along each branch before backtracking to the parent. In order to show the list of paths and the different changes of each node in the path, we have created a tree component, where each tree path represents the sequence of nodes(files) to be tested by the testers of the application. Each node has sub-nodes that classify the changes into three types. On the front side and to show the changes in the files as categories, the framework used the react D3 Tree component to represent hierarchical data.

### *Evaluation of RN-AST Pruning Framework*

A user evaluation was conducted to evaluate the user experience and acceptance and measure the effectiveness of the framework in detecting and finding the UI differences between the original and updated source codes.

The evaluation was done on a React Native application made for testing. This application is made of a list of pages, each page consists of one or more core components. These pages are treated as modules. Thus, importing one module into another module produces the dependency between these modules.

### *Participants*

Basically, the evaluation of the proposed framework was done with the help of six volunteers. Two of them are experienced react Native testers and the others are novice testers in react Native, but a comprehensive tutorial was given to them in order to teach them the basics and increase their knowledge of using and programming with react Native. In general, participants interact with the RN-AST pruning framework using a web interface implemented using the react library. Results and errors that were generated on the server were sent to the client and displayed to them. The output sent between the server and the client side was implemented as a JSON object as many web applications use this format for data transmission.

### *Experiment Procedure*

Below are the steps that were followed to help in evaluating the RN-AST pruning framework.

The participants were introduced to the framework by reading an instructional tutorial document to reduce the bias between the different participants, and to demonstrate the main aim of this framework and the steps of using it with an explanation of the framework's outputs.

Multiple online sessions were done with the participants to introduce them to the tool using the Zoom application.

On completion of the learning step, the participants were asked to make some GUI changes to an application under use, these changes included adding new elements and deleting or updating existing elements.

The changes done by the participants were run to ensure that there were no run-time issues

Their changes were cloned and copied to the proposed framework.

RN-AST pruning framework starts its job by calling the Madge API to build the dependency graph of the original source code, produce the AST by parsing the different versions of the source code using the babel parser, prune the AST to keep only the GUI expressions, and finally compare the pruned AST's to check if the file has changed before building the paths that contain the files with changes.

Upon finishing the experiment showing the results to the participants and making sure that the tool detects their changes, a matching questionnaire was filled with each participant.

### *The Questionnaire*

In our study, the questionnaire method was used to collect, obtain, and summarize useful information from the participants about the proposed framework to support and help the evaluation process.

The questionnaire used in this study was made with the help of Google form and it followed the positive design approach, which was introduced. It suggests including items with positive and negative wordings to reduce the response biases, help analyze the results faster, and avoid accidental errors. The questionnaire has three sections with a total of 16 questions, 9 questions with 5 point Likert-scale ranging from strongly agree to strongly disagree, 3 open-ended questions to capture their opinions of using RN-AST pruning framework in testing react Native applications, and the remaining questions to capture their background and experience in developing mobile applications.

## Results and Discussion

The participants varied in terms of their highest qualifications and in their experience in mobile development. However, they were able to use the framework, make code changes, and get the affected GUI elements and paths that need to be tested due to the code changes between the original and the updated version of the source code. This ensures that RN-AST framework can effectively be used by test engineers regardless of the experience level, the highest qualifications, and making changes to the original source code as illustrated in react Native skills.

### *Participants Experience*

According to their experience in the mobile development field, results as illustrated in Fig. 2 indicate that only one of them is senior with more than 6 years in this field, one of them has no experience and the other participants are juniors with only 1-3 years experience.

According to their familiarity with React native, one of them is not familiar with this framework and the other has heard about it and has little experience in this field. All testers were able to understand the usability of the framework and add, delete, or update the application GUI elements. Moreover, all of them were able to get the list of affected elements and files that need to be tested to cover the code changes.

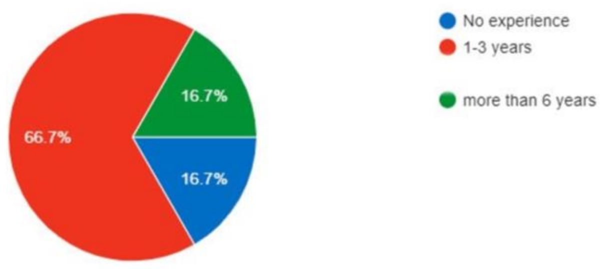

**Fig. 2:** Mobile development experience

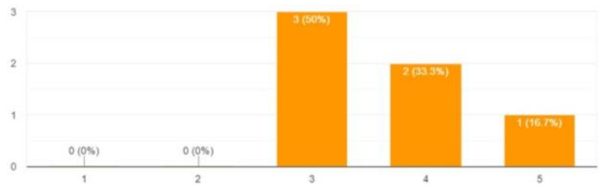

**Fig. 3:** Easy-to-use RN-AST pruning framework

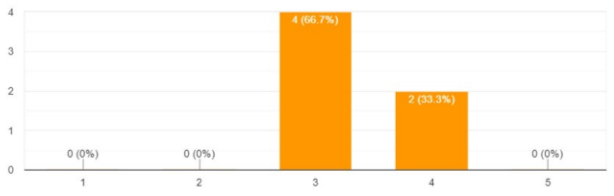

**Fig. 4:** Easy to make changes on the original source code

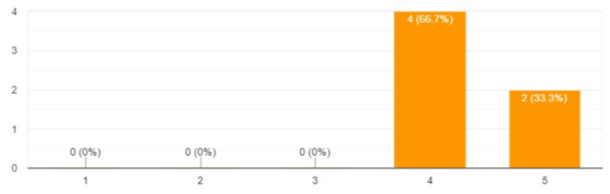

**Fig. 5:** Provide the affected files

### *Usability and Efficiency*

The second section of the questionnaire aims to evaluate the usability of using the RN-AST Pruning framework. As indicated in Fig. 3, participants varied in determining the usability of the RN-AST Pruning framework, half of them were neutral and the other half found it easy to use.

In addition, two participants found it easy to make code changes and the other was neutral in determining the ease of making changes to the original source code as illustrated in Fig. 4.

As for the efficiency of the RN-AST Pruning framework, almost all participants agreed that the proposed framework was able to provide them with the affected files that need to be tested as shown in Fig. 5, where 5 in Likert-scale means totally agree.

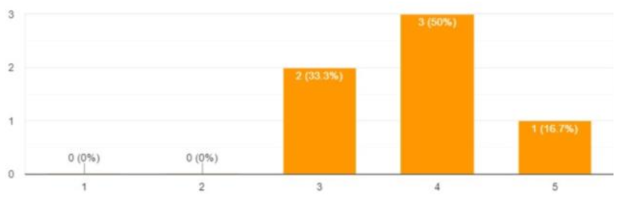

**Fig. 6:** Satisfy with the results

The results provided by the RN-AST Pruning framework as illustrated in Fig. 6 were satisfying for more the half of the participants and the other participants were neutral in determining their satisfaction with the framework results.

Despite all, almost all participants believe that the idea behind this framework is useful and worthy. After collecting the results from the participants, below points can be shown:

- All participants agreed that the RN-AST Pruning framework provides them with the list of affected files. However, two participants were not satisfied with the way the results were shown and they found it hard to understand
- The RN-AST Pruning framework idea is useful and can help test engineers by reducing the time and effort required to do the testing process by providing them with the affected files and paths

From these two considerations the fourth research question "How effective is the build framework in detecting changes and results that satisfy the test engineers?" can be answered positively to indicate the efficiency of the RN-AST Pruning framework idea and implementation.

### *Threats to Validity*

Although we did our best to reduce threats to the validity of the experiment, there are certain threats faced while implementing the framework.

RN-AST Pruning framework has been tested and evaluated by a small sample of React Native testers and developers. However, in order to obtain more accurate results, the RN-AST Pruning framework has to be tested by a larger sample to cover a large area of GUI changes and more react native components. In general, resource constraints limit the ability to collect data at a reasonable cost. In our study, time and money were two resource limitations that directly influenced how much data could be collected, as it was difficult to find React native programmers and test engineers to help in testing our framework for free and in a specific period of time. Therefore, this can justify the small sample size in our research. The resource constraints and limitations also affect the covered components. RN-AST Pruning framework only covered the core components in React Native, but the increase in using mobile applications and the diversity in application domains may possibly reveal other components to be included in the framework.

## Conclusion

In this study, the RN-AST Pruning framework was introduced. The basic idea behind this framework is to enhance the testing process and help test engineers by reducing the number of test cases to run, therefore, reducing the required time and effort needed to complete the testing process. The framework works on pruning the abstract syntax tree generated by code static analysis to keep only the GUI elements. Then compare the pruned AST of the original source code with the pruned AST of the updated version of the source code. GUI changes are categorized into updates, placements, and deletions. The framework then builds the paths that contain the changed files. Each path has a list of changed files or the files that may affected due to the dependency with that file. Thus, reducing the number of tested files and the number of test cases to run. The framework was evaluated using a case study evaluation conducted on a group of six developers with different qualifications and experiences. Participants used the RN-AST Pruning framework to detect their changes on the proof-of-concept mobile application and to list the affected files that need to be tested for them. Results show that the framework was able to provide them with the changes they have applied to the mobile application code. Moreover, they praised the framework and believe that it is useful in helping test engineers in their testing process.

Due to the lack of time and to improve this study, some next steps need to be conducted in the near future:

- Increase the number of covered components
- Support the conditional rendering
- Enhance the framework to a more user-friendly interface
- Evaluate the framework with the help of a larger sample of developers and testers
- Integrate our results with model-based GUI test case generation tools to get more accurate and systematic results

## Acknowledgment

I would like to thank my supervisor Dr. Samer Zein for his expertise, which helped in completing this study successfully. Thanks also go to the volunteers for their dedication and commitment which was critical in achieving our research goals.

## Funding Information

The authors confirmed that there is no funding agency exists in this study.

## Author's Contributions

**Rand Marwan Khalil Ibrahim:** The study background and related work, methodology, programming, implementation, data collection, and experiments.

**Samer Zein:** Research Idea, reviewed the manuscript, audit identification, and audit tracking.

## Ethics

This manuscript is original and contains unpublished material. The authors conducted their research ethically, following the ethical principles and guidelines of their field and institution.

## References

Bauersfeld, S. (2013, March). GUIdiff--A Regression Testing Tool for Graphical User Interfaces. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation* (pp. 499-500). IEEE. https://doi.org/10.1109/icst.2013.84

Eisenman, B. (2015). *Learning react native: Building native mobile apps with JavaScript*. "O'Reilly Media, Inc". ISBN-10: 1491929073.

Hartmann, G., Stead, G., & DeGani, A. (2011). Cross-platform mobile development. *Mobile Learning Environment, Cambridge*, *16*(9), 158-171.

Memon, A. M. (2002). GUI testing: Pitfalls and process. *Computer*, *35*(08), 87-88. https://doi.org/10.1109/MC.2002.1023795

Raj, C. R., & Tolety, S. B. (2012, December). A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)* (pp. 625-629). IEEE. https://doi.org/10.1109/indcon.2012.6420693

Salihu, I. A., Ibrahim, R., & Mustapha, A. (2017). A hybrid approach for reverse engineering GUI model from android apps for automated testing. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, *9*(3-3), 45-49. https://jtec.utem.edu.my/jtec/article/view/2870

Tao, C., & Gao, J. (2016). On Building Test Automation System for Mobile Applications Using GUI Ripping. In *SEKE* (pp. 480-485). https://www.ksiresearch.org/seke/seke16paper/seke16paper_168.pdf

Utkin, I., Spirin, E., Bogomolov, E., & Bryksin, T. (2022). Evaluating the Impact of Source Code Parsers on ML4SE Models. *arXiv preprint arXiv:2206.08713*. https://doi.org/10.48550/arXiv.2206.08713

Yang, W., Prasad, M. R., & Xie, T. (2013, March). A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 250-265). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-37057-1_19