An Empirical Study of Regression Testing for Android Apps in Continuous Integration Environment

Dingbang Wang
University of Cincinnati
Ohio, USA
wangdb@mail.uc.edu

Yu Zhao
University of Central Missouri
Missouri, USA
yzhao@ucmo.edu

Lu Xiao
Stevens Institute of Technology
New Jersey, USA
lxiao6@stevens.edu

Tingting Yu
University of Cincinnati
Ohio, USA
tingting.yu@uc.edu

***Abstract*—Continuous integration (CI) has become a popular method for automating code changes, testing, and software project delivery. However, sufficient testing prior to code submission is crucial to prevent build breaks. Additionally, testing must provide developers with quick feedback on code changes, which requires fast testing times. While regression test selection (RTS) has been studied to improve the cost-effectiveness of regression testing for lower-level tests (i.e., unit tests), it has not been applied to the testing of user interfaces (UI) in application domains such as mobile apps. UI testing at the UI level requires different techniques such as impact analysis and automated test execution. In this paper, we examine the use of RTS in CI settings for UI testing across various open-source mobile apps. Our analysis focuses on using Frequency Analysis to understand the need for RTS, Cost Analysis to evaluate the cost of impact analysis and test case selection algorithms, and Test Reuse Analysis to determine the reusability of UI test sequences for automation. The insights from this study will guide practitioners and researchers in developing advanced RTS techniques that can be adapted to CI environments for mobile apps.**

***Index Terms*—Regression testing, Android apps, Empirical study**

I. Introduction

Continuous integration (CI) has been widely used in practice to rapidly integrate code changes, new features, test execution, and provide feedbacks. When changes are committed to the project hosted by the CI system (e.g., Jenkins, Travis), tests are automatically executed to check whether the changes break the functionality of the code. A survey conducted on more than 400 software developers suggest that about 70% developers found that CI is able to help them catch bug earlier and make them less worried about breaking the builds [1].

In order to provide fast feedback for developers, it is important to run test cases as quickly as possible. Regression test selection (RTS) techniques have been employed to focus on running only tests that are relevant to the changes [2]. Traditional RTS approaches often cannot be adopted in CI environments primarily because they perform fine-grained analysis (e.g., code instrumentation) to select test cases and thus require significant analysis time. Recent work has quantitatively studied to what extent RTS is needed in CI environments and the cost-effectiveness of RTS techniques at different levels of regularities [3]

Most existing techniques and studies on CI testing focus on lower-level tests (e.g., unit tests) [1]. This is because unit tests are often generated in early development process and can execute quickly to provide fast feedback. However, unit tests are not able to detect user-level runtime failures. Some application domains, such as mobile apps, have rich graphical user interfaces (GUI). Therefore, testing the GUI of these apps is critical and a lot of research has been focusing on this topic [4]. When a change happens, regression testing is still needed to validate the changes will not adversely affect the GUI of the application. There are a few approaches on developing regression testing techniques, such as RTS, to efficiently validate the effects of changes on the application UI [5]–[7]. For example, QADroid [5] is one of the first approaches to apply RTS on mobile apps. Specifically, QADroid performs an event-level impact static analysis to select the GUI events affected by the app changes.

However, there are still many open challenges that have not been identified or addressed in existing research. For example, while there have been many approaches on studying CI testing [3], [8] or designing new RTS techniques to support CI testing, none of them have focused on testing the GUI of mobile apps. It is worthwhile studying the cost and effectiveness of CI-based RTS for mobile apps. Our empirical study indicates that out of the 318 open-source apps hosted on GitHub that we selected from the Google Play Store, 40.25% of them employ CI. In addition, existing RTS techniques for mobile apps focus on selecting affected events, which are not directly executable and require manual testing. Therefore, it is necessary to study how to automatically execute the test sequences relevant to the affected events.

The goal of this study is to identify opportunities in performing CI testing for mobile apps at the UI level to understand the cost and effectiveness of regression testing. We perform an in-depth study on 473,225 commits of 318 open-source apps selected from the Google Play Store. For different apps, we begun with analyzing the frequency of commits on GitHub and the frequency of functional changes only because not all the changes are related to the logic of UI, such as commits of code refactoring and documentation. Our results suggest that a significant proportion of functional commits in Android UI testing occur within a short timeframe of 30 minutes, indicating the need for RTS. However, the majority of functional group commits occur in time intervals greater than two hours, suggesting that when considering

978-1-6654-5223-6/23$31.00

testing groups of commits made by the same developers, it is unlikely that RTS will be necessary, potentially leading to more efficient development processes.

In addition to code change frequency, we also analyze the cost of performing program analysis for identifying affected UI and the cost of test sequence execution. The goal is to assess the cost of UI-level RTS when dealing with the speed of changes. Specifically, we use dynamic analysis to identify the UI affected by the code changes and compute the cost. We then execute the test sequences including the affected UI. The cost of executing the test sequences containing the affected UI will allow us to estimate the time needed to test the UI affected by the changes. Our results suggest that while the cost of creating the initial test sequences can be high, selecting test sequences containing affected UI elements can significantly reduce the cost of testing.

Finally, we study the re-usability of test sequences in the presence of changes. We study to what extent the the app can reuse existing test sequences for the changes. We study the change-prone UI by analyzing the frequency of each UI being changed across commits. We hypothesize that the change-prone UIs are more likely to be the targets and thus automatically selected for testing. Our results indicate that test sequences that already exist are often highly reusable. In addition, there are certain UI elements commonly observed across changes in some applications; prioritizing testing of these UI elements without extensive analysis could be more efficient.

II. Background and Motivation

In this section, we describe background on regression test selection techniques and their application on mobile apps. We then describe the background of testing in CI environments.

A. Background

Regression Test Selection for Mobile Apps. Let P be a program, and let P' be a modified version of P. Regression Test Selection (RTS) techniques aim to reuse existing test cases in P by selecting a subset of test cases that are important to rerun, omitting less important test cases. Several approaches have been proposed for rendering reuse more cost-effective via regression test selection [9]–[14].

However, most existing RTS techniques focus on traditional applications and cannot be directly applied to testing the user interface (UI) of mobile apps, which are typically event-driven and have different system architectures. To test the UI of mobile apps, an impact analysis must be able to identify changes at the UI level rather than the code level, as in traditional approaches, so that certain UI components will be selected to test the app functionality affected by the changes.

While a significant amount of work has been proposed to test mobile apps [15]–[23], little has focused on studying or developing RTS for them. To the best of our knowledge, QADroid [5] is the only approach that can perform event-aware RTS for testing the UI of Android apps. Specifically, QADroid develops an event-level impact analysis technique to determine which UI events are affected by the app changes, so that developers can focus on creating test sequences focusing on these events.

Continuous Integration (CI) Testing. Continuous Integration (CI) encourages developers to break their work into smaller pieces and commit changes to the repository frequently. This enables them to track changes and receive fast feedback if any tests fail due to the changes. To maintain a working software product, tests are executed whenever a change occurs to examine if the change breaks the build. Therefore, the affordability of test selection is determined by the frequency of changes and the cost of exercising the tests.

However, traditional Regression Testing Selection (RTS) techniques cannot be applied in CI environments due to their typically long analysis time involving certain testing-related tasks such as impact analysis and collecting coverage. Recent research proposes to make regression testing more cost-effective in modern software projects [10], [24]. For example, Gligoric et al. [24] propose a class-level dynamic RTS technique that is more efficient than finer-level dynamic RTS, as shown by their evaluation. Elbaum et al. [25] create RTS techniques that use time windows to track how recently test suites have been executed and revealed failures.

Hilton et al. [26] study the usage of CI in open-source projects, such as to what extent CI is adopted in software development. Legunsen et al. [9] evaluate the performance benefits of static RTS techniques and their safety in modern software projects. Memon et al. [27] share their experience and results in RTS on Google projects. Vasilescu et al. [28] study the productivity of CI based on 246 GitHub projects. However, none of the existing CI testing techniques or studies have targeted mobile apps.

GUI testing. In mobile and web applications, a GUI widget refers to a graphical element, such as a button, text field, or check box, used in the app's interface. A generic event refers to any action that is triggered by the system or the user, while a GUI event is an executable GUI widget associated with a particular event type, such as click, long-click, swipe, or edit.

GUI testing is the process of generating a sequence of GUI events with the goal of achieving optimal code coverage and detecting bugs. By testing the app's graphical interface, developers can ensure that the app's functionality is working as intended and that the user experience is smooth and intuitive.

B. Motivation

Although recent research has considered applying RTS to perform regression testing for mobile apps, challenges still remain concerning its cost-effectiveness. First, the cost of selecting the affected UI depends on how the impact analysis was done, which directly affects the applicability of RTS in fast-changing environments, such as CI. While the practicality of RTS in CI has been studied [9], [26]–[28], none of them has considered the scenarios of event selection for mobile apps.

Second, existing research, such as QADroid [5], focuses on event selection, whereas another key phase contributing to the cost is executing the events. Unlike traditional unit testing for which tests are directly executed by the test engine (e.g., JUnit framework), UI testing of mobile apps needs to exercise sequences of events. That being said, sequences that include the affected events should be automatically selected and executed by an RTS approach. To the best of our knowledge, none of the existing techniques can achieve this goal.

Therefore, this study will investigate whether and how RTS can be used for testing mobile apps in the presence of speedy app evolution, such as CI environments. We expect the results to provide insights and guidance for designing cost-effective RTS techniques for mobile apps. Our study will be centered around the aforementioned challenges. Specifically, we will first study how frequently mobile apps are changed to provide an overview of the app evolution and the necessity of using CI-based RTS. Second, we will study the cost of RTS with the current state-of-the-art impact analysis and event execution. We expect the results to demonstrate the practical aspect of performing RTS in CI environments for mobile apps (Challenge 1). Finally, we will study the re-usability of test sequences to assess the feasibility of automating the execution of test sequences (Challenge 2).

In summary, our study aims to address the challenges that still remain in applying RTS for mobile app testing. By investigating the cost-effectiveness of RTS in fast-changing environments, such as CI, and exploring the re-usability of test sequences, we expect to provide valuable insights for designing effective RTS techniques for mobile apps.

III. Experiment Design

A. Development of Research Questions

Our goal is to understand the key factors that affect the applicability of regression test selection techniques for mobile apps in CI environments. To this end, we design our research questions in three dimensions: frequency analysis, cost analysis, and test reuse analysis.

1) Frequency Analysis.: The frequency of the changes would play an important role to analyze the cost of regression test selection for mobile apps in CI environments. Existing research has shown that traditional RTS is not suitable for CI environment with fast change speed because it often requires a significant amount of time to perform the analysis to select an accurate set of tests. The main concern of applying traditional testing techniques in CI is that the analysis and testing time can be too much to catch up with the speed of changes [25]. Therefore, the degree to which RTS can be applied depends on the arrival rates of commits or commit intervals (i.e., the time between consecutive commits). For example, if changes happen frequently in short intervals, the efficiency of RTS is very important in order to minimize analysis and testing times. On the other hand, if changes usually happen in larger intervals (i.e., low commit arrival rates), such as when the project reaches a certain development period, it might be okay to just re-execute all test sequences to simplify the testing process.

Prior work [3] also demonstrated that a 60% of open-source projects do not even need RTS because their average commit interval is smaller than the time of executing all tests. Therefore, we study the frequency of commits. Therefore, we first ask the following research questions: **RQ1:** *What is the frequency of single commits?* In this RQ, we analyze the frequency of overall commits, also focusing on the distinction of the frequencies between functional and non-functional commits. The insight is that some changes are not directly affecting the behavior or functionality of the application, which are not related to UI logic, which suggests that there is no need to execute all existing UI test sequences for non-functional changes commit. This insight has important implications for optimizing the UI testing process and reducing unnecessary resource consumption. Also, Developers tend to make multiple commits within a short time timeframe and they may prefer to run tests after multiple commits instead of after each individual one. Hence, we formulate **RQ2:** *What is the frequency of group commits?*

Answering the above RQs would assess if CI is needed in Android development community. If the functional change frequency is low, then maybe traditional testing is sufficient.

2) Cost Analysis: The cost of RTS often includes the time needed for analysis (e.g., impact analysis, test selection algorithms) and test execution. The cost is critical in CI testing because a very long testing time defeats the purpose of having developers receive fast feedback. This motivates us to study the cost of RTS in mobile apps. However, the challenge is that UI testing is different from traditional value-based testing (i.e., providing an input and checking the output) in several aspects. First, the impact analysis is performed at the event level, so the analysis needs to associate the code with particular UI events. Therefore, we answer the following research question: **RQ3:** *What is the cost of program analysis for identifying affected events?*

Second, in traditional unit testing, tests are selected if their associated code is changed or affected. However, UI testing is at the level of event sequences. Even if the affected events are selected, one still needs to search for event sequences to exercise the affected events for automated testing. Therefore, we would like to examine the cost of finding and exercising such event sequences. Hence, we ask in **RQ4:** *What is the cost of exercising the affected events and how does this cost relate to the time intervals between commits??*

3) Test Reuse Analysis: It can be costly to dynamically search for testing sequences. Research has shown that a targeted search can take anywhere from 16 seconds to over two hours, with an average of about 6 minutes [29]. A more practical approach is to reuse existing test sequences that contain the affected events. However, it may not always be possible to reuse existing sequences when events are added or deleted. Thus, we aim to study the feasibility of reusing existing test sequences during test selection. Therefore, we pose the following research question: **RQ5:** *What test executable rate of executing existing test sequences from the initial commit version to subsequent commit versions? To what extent existing test sequences can be reused across changes?*

Some traditional regression test selection and test prioritization techniques examine the history of changed code or source files to determine the tests to be selected or prioritized. The insight is that frequently changed code is more prone to bugs than other parts of the code. Thus, the tests associated with the change-prone code are selected for continuous integration (CI) testing. In the context of UI testing, we can examine the frequency of changed events so that frequently changed events can be selected for testing. Therefore, we ask the following research question: **RQ6:** *What is the percentage of affected events that are commonly seen across changes*

B. Data Collection

Project selection. We collect open-source projects from the Google Play Store by utilizing web crawling techniques [30].

By iterating over the web pages of more than 100,000 apps, we successfully identified approximately 1,000 open-source Android mobile apps by filtering keywords related to open-source projects in their web pages, such as $open-source$, $opensource$, $GitHub$, $GitLab$, etc. Among these, 318 host their code repositories on GitHub, providing a valuable resource for further analysis and insights into the development process of Android applications. We based our study on 318 applications and employed web crawling techniques [30] again to collect detail commit information from each commit page, such as the code changes (both additions and deletions), methods, classes, and etc.

To ensure that the study is conducted under a consistent and automated setup, we identified and selected projects that could be successfully built using the Gradle command line build. This choice allows us to automatically checkout a series of continuous commit IDs, connect an emulator, build and deploy the corresponding commit version of APK, and execute test sequences just by utilizing bash scripts and command lines. We selected 100 continuous commit versions only for each project to strike a balance between the depth of analysis and the feasibility of conducting the study within the given time and resource constraints. We made the decision not to exclude non-functional commits to simulate real-world scenarios in a continuous integration environment. In such environments, developers typically do not have prior knowledge about whether a commit is functional or not.

Table. I shows the characteristics of the overall of projects and projects selected for studying RQ3-6, including details on the number of projects, total commits, size, age distribution, and the usage of CI/CD. The usage of CI/CD suggests the popularity of CI in the Android development community. In order to determine if a GitHub project uses CI/CD, one can search for common files such as .travis.yml or .github/workflows/, or checking for build status badges in GitHub.

	Overall Projects	**Projects(RQ3 - RQ6)**
# Projects	318	20
Total Commits	473,225	1475
Size (LOC)	Min: 406 Median: 17,612.5 Max: 2,211,640	Min: 4,026 Median: 34,681.5 Max: 114,760
Ages (yrs)	Min: 0.17 Max: 14.5	Min: 0.85 Max: 14.5
CI/CD Usage	Total: 40.25% Travis CI: 28% GitHub Workflows: 24.53%	Total: 85% Travis CI: 85% GitHub Workflows:80%

TABLE I: Overall Projects vs. Projects for RQ3-6

C. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our subjects and the selected commits (RQ3–RQ6). The data we gathered comes from 40.25% of projects using CI/CD with 28% deployed in Travis CI environments and 24.53% using GitHub Workflows. Other subjects and CI servers may exhibit different behaviors. However, we do reduce this threat to some extent by systematically searching for open-source code subjects from Google Play Store for our study. A second source of potential threat involves the selection of commits used to study the cost of analysis. Since we selected the first 100 commits from the commits version that could be built using the Gradle command line build, they may not represent the all commits over the commit history.

The primary threat to internal validity for this study is the potential occurrence of errors during the implementation of dependency analysis in RTS. To address this concern, we took measures to control the threat by conducting extensive testing of our tools. We also validated their results against a smaller program for which we could manually determine the correct outcomes. Secondly, we employed dynamic analysis to identify the affected events and test sequences. However, it is important to note that dynamic analysis might not cover all the affected code if it is not executed. To overcome this limitation, future studies will incorporate both dynamic and static analysis to provide a more comprehensive perspective. Thirdly, we utilized a depth-first search algorithm to generate test sequences, which is a widely adopted approach [18], [29], [31]. However, future studies could benefit from employing state-of-the-art tools [32]–[35] to generate test sequences, enabling more thorough insights into the subject matter.

The primary threat to construct validity involves the dataset and metrics used in the study. To mitigate this threat, we used open source projects from GitHub, which are publicly available and well studied by existing research [28]. We have also used well known metrics in our data analysis such as the number of commits and correlation analysis, which are straightforward to compute.

IV. Results

A. Frequency Analysis

The first research question (RQ1) aims to investigate the frequency of code commits in Android UI testing and determine the extent to which RTS (Regression Test Selection) is required. The purpose of this research question is to identify the circumstances under which RTS is necessary and those in which it may not be required. For example, in projects with low code change frequency, RTS may not be needed. Conversely, in projects with multiple active authors, code commits may occur in a rapid succession. In such cases, executing tests after a group of commits rather than after each individual commit could be more cost-effective. Therefore, the study conducts commit frequency analysis for both single commits and group commits.

1) RQ1: What is the frequency of single commits: In Android UI testing, a single code commit can be classified into two types: functional and non-functional changes. Functional changes directly affect the application's behavior, features, or capabilities, while non-functional changes relate to aspects such as maintainability, readability, or performance, which do not directly application's behavior most of the time. As noted in the Data Collection Section, we used web crawling techniques to obtain details of each commit page. This allowed us to examine a vast number of commits to identify if the commit is functional or non-functional by using heuristics code to examine if there are any additions and deletions of calling functions.

Functional commits are more likely to require testing than non-functional commits since they can affect the software's behavior. To understand the extent to which RTS is necessary,

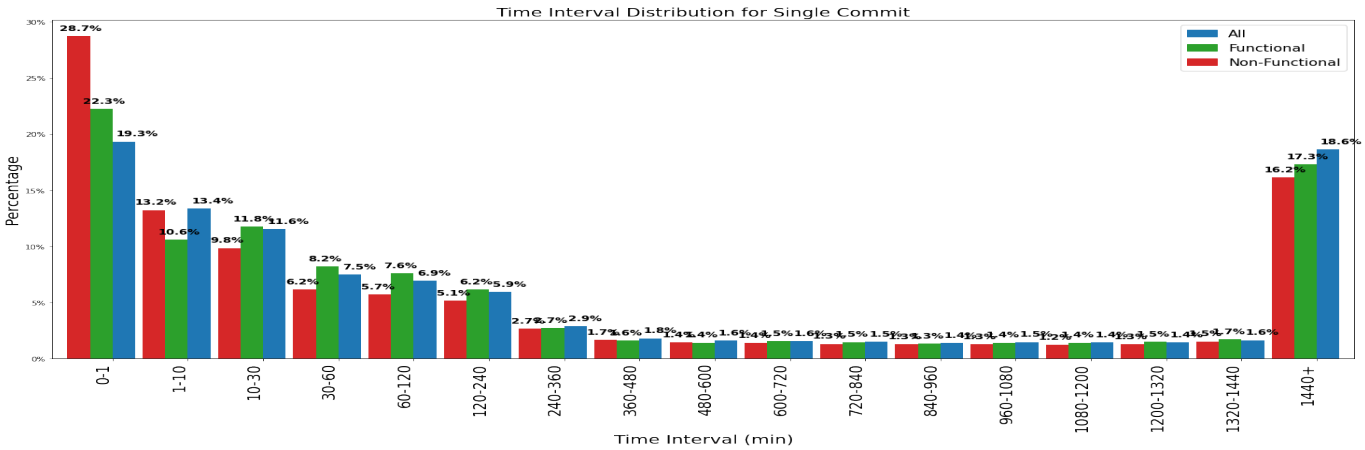

Fig. 1: Time Interval Distribution for Single Commits

it is crucial to distinguish functional commits from other types of commits.

Our study analyzed 473,225 commits from 318 apps, where 187,881 (39.7%) were functional commits and 285,344 (60.3%) were non-functional. Figure 1 displays the frequency distribution of all commits, functional commits, and non-functional commits. The x-axis represents the range of time intervals between two consecutive commits, binned into different ranges from 0-1 minute to 1440+ minutes. The y-axis shows the percentage of commits that fall into each interval range. The figure shows that overall commits, functional commits, and non-functional commits have a similar trend, with the percentage of commits decreasing as the time gap between consecutive commits increases. This trend indicates that functional commit frequencies are not significantly different from overall commit frequencies in terms of their time distribution.

Our analysis revealed that a significant proportion of functional commits in our dataset occurred within a relatively short time period. Specifically, 44.7% of functional commits were observed to occur within 30 minutes, indicating that a large proportion of functional changes are introduced into the codebase shortly after the previous commit. Within 60 minutes, this percentage increased to 52.9%, indicating that the majority of functional changes occur within the first hour of committing code changes. Additionally, our analysis found that a slightly higher percentage of functional commits (60.5%) occurred within 2 hours of the previous commit. Many existing techniques, such as those proposed in the literature [22], [34], [35], typically allocate a threshold of 1-3 hours to complete a thorough GUI testing.

RQ1: A considerable number of functional commits occur within a short timeframe of 30 minutes. This rapid pace of development indicates that the development team is making changes to the codebase frequently and possibly in a highly iterative manner. Therefore, it is essential to ensure that testing strategies are efficient enough to keep pace with the speed of development. Specifically, our results imply that if the execution time for testing exceeds 30 minutes, it may be necessary to use Regression Test Selection (RTS) techniques to optimize the testing process.

2) RQ2: What is the frequency of group commits?: App development is generally a collaborative effort, involving multiple developers who work together to maintain and enhance the project. Our data indicates that the 5-number summary of the number of developers of studied apps includes a minimum value of 1, a first quartile value of 2, a median value of 5, a third quartile value of 21, and a maximum value of 1983. Developers tend to make multiple commits within a short time period and they may prefer to run tests after multiple commits rather than after every single commit.

This observation leads to our thoughts to consider an enhancement for RTS: A single commit can be seen as a test event, and a group of commits is also considered a test event. Can testing efficiency be improved by running tests on a per-group basis rather than per single commit basis? A *Group commit* refers to a sequence of one or more consecutive commits made by the same author within the same date. Our data, drawn from 473,225 single commits, indicates that 154,124 groups were formed by the definition above, representing a reduction of 67.47% in the number of test events. Of these groups, 46.64% of groups were found to be functional, while 53.36% of groups were considered non-functional. A group of commits can be considered functional if it contains at least one functional commit.

The results indicate a reduction of 67.47 in the number of test events from 473,225 single commits to 154,124 group commits. The reduction is 61.85% when considering functional group commits. With grouping commits, the number of test events is significantly reduced, meaning that less testing is needed.

Fig.2 (a) displays the time interval distribution of group commits, demonstrating a consistent trend among all group commits, functional group commits, and non-functional group commits. The graph reveals that less than 20% interval falls in [0,30], especially for functional group commits, only 10% of the intervals fall within the [0, 30], while the majority are distributed across the [120, 1400) and [1400, inf) intervals. This observation suggests that there are considerable time gaps between group commits, providing ample time for testing.

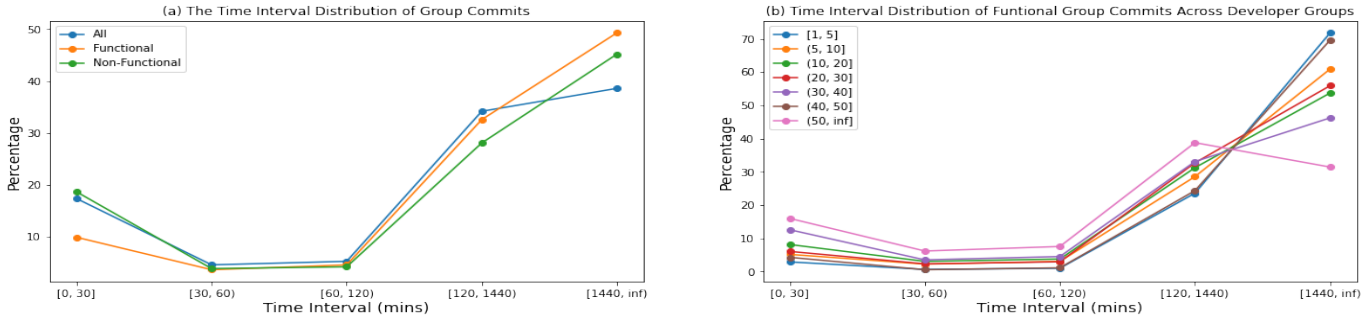

Fig. 2: Time Interval Distribution for Group Commits

Consequently, there may be no need the introduce the idea of group commits for RTS for UI testing.

However, it is essential to consider the potential bias due to the wide range of numbers of developers in different apps, as indicated by the 5-number summary of the number of developers (1, 2, 5, 21, 1983). To account for this possible bias, the data has been divided based on the number of unique developers in each app, forming groups [1, 5], (5, 10], (10, 20], (20, 30], (30, 40], (40, 50], and (50, inf] for functional group commits. The range of these groups is determined by the distribution of unique developers of studied apps. Fig.2 (b) shows the time interval distribution of functional group commits across developer groups. While the trends remain similar to those seen in Fig.2 (a), it is observed that the largest two developer groups, (40, 50] and (50, inf), exhibit the highest percentage in the (0, 30] interval and the lowest percentage in the (1440, inf) interval. In contrast, the smallest developer group [1,5] has the lowest percentage in the (0, 30] range and the highest percentage in the (1440, inf) range. Nonetheless, the highest percentage in the (0, 30] interval remains small, less than 20%.

This observation suggests that although the higher number of developers does contribute to an increase in the percentage of functional group commits in smaller time intervals, the increase is still relatively minimal. As a result, the need for the idea of group commit strategy may not be significant, even for applications with a higher number of developers.

RQ2: We found that a relatively small percentage of functional group commits (9.9%) were made within time intervals of less than 30 minutes. The majority of functional group commits (81.83%) occurred in time intervals greater than two hours. Specifically, 32.57% of the functional group commits fell within the time range of [2, 24) hours, while 49.26% exceed 24 hours. These findings suggest that when considering testing groups of commits made by the same developers, it is unlikely that RTS will be necessary. Therefore, this information can guide developers in optimizing their testing strategies and prioritizing their testing efforts, potentially leading to more efficient development processes.

B. Cost Analysis

1) RQ3: What is the cost of program analysis for identifying affected events?: Given a code change c, the analysis will identify a set of GUI widgets W associated with the code that is affected by the change c. The GUI widget $w \in W$ is considered affected GUI. In order to identify the affected GUI, we needed to obtain the mapping between GUI widgets and the code. When the source code has changed, the GUI related to the changed method can be quickly found with the help of mapping information. There are two general approaches to obtaining the mapping information from call graph to identify the affected GUI: static analysis and dynamic analysis.

In this study, we chose dynamic analysis as the preferred approach for establishing the mapping between UI elements and their corresponding methods. The rationale behind this decision lies in the ability of dynamic analysis to provide a more accurate and straightforward representation of the application's runtime behavior. By monitoring user interactions with the UI and logging the invoked methods, dynamic analysis allows us to directly observe the relationships between UI elements, their associated event handlers, and methods called inside handlers, effectively circumventing the need for iterating through the call graph. This, in turn, eliminates the consideration of unnecessary paths and potential inaccuracies that might arise during static analysis. For example, when identifying the method related to a button, the static analysis might require examining the entire codebase, the associated resource references of the button, constructing the static call graph, traversing the call graph of find all methods containing the associated resource references of the button, and potentially considering irrelevant paths or methods. In contrast, dynamic analysis involves monitoring the actual user interaction, while clicking the button, information of invoked methods would be logged, which allows direct establishes of the relationship between the button and its related methods.

While the dynamic analysis may require more time due to code execution, its ability to capture the precise execution paths and sequence of method calls outweighs these drawbacks. In fact, constructing and iterating the entire static call graph can be a time-consuming process. Consequently, the dynamic analysis presents itself as a superior method for mapping UI elements to their related methods, delivering a more reliable and accurate understanding of the application's behavior, particularly regarding user interaction and event handling.

Generating test sequences for initial app version. When performing RTS, it is necessary to have initial test sequences to choose from. However, many of the apps in our dataset do not have default initial test sequences. To address this challenge, we developed our own tool that generates initial test sequences. This tool employs the Depth-First Search (DFS) algorithm, as utilized in existing work [18], to automatically explore the app. The DFS algorithm traverses as far as possible along each GUI sequence before backtracking Note that in our study, the term "backtracking" does not refer to pressing the "back" button or performing any stepping back actions. Instead, it refers to restarting the application and re-executing the sequence from the initial point up to the decision point. For example, by clicking on the menu on the home page, the instrumented source code will execute the methods under the action of clicking, along with the resource ID of the GUI element involved. Utilizing the collected information about GUI elements, a series of executable GUI test sequences are generated. The number and length of test sequences can be influenced by the complexity of the app's interface and the actions required to navigate to the next state.

Our initial test sequences contain *various numbers of sequences ranging from 23 to 123, with the longest sequence in each set varying between 4 and 13*. The process of collecting sequences takes a variable amount of time, ranging from 32 to 150 minutes, in the study. We have employed two stopping criteria maximum sequence and length and time limit. When the search reaches the depth of 15, it backtracks and continues exploring other possible sequences and we imposed a time limit of 150 minutes to ensure the algorithm does not run indefinitely. It mainly includes the time spent on analyzing the hierarchy of the current screen to identify the candidates for the next step, locating UI elements, performing operations on them, restarting apps and navigating to the previous state for every exploration. Note that achieving comprehensive code coverage is extremely challenging [18], [22], [32], [33], [35], and dynamic DFS does not serve as the optimal solution. In this work, we only required basic sequences and thus do not need to concentrate on generating complex sequences or sequence combinations to enhance code coverage.

Test sequence selection. Our approach to RTS involves establishing a mapping between methods and associated GUI widgets, denoted as $< m, W >$, where m is a method and W is the set of GUI widgets associated with that method. This mapping allows us to quickly identify the affected GUI widgets W when a code change c occurs. By locating the corresponding method m, we can obtain the set of associated widgets W, and any test sequences containing $w \in W$ can be selected for execution.

The efficiency of our approach is supported by the constant time complexity of O(1) for identifying the affected GUI widgets W. This is due to the fast execution of the map.get() function, with retrieval *times ranging from approximately 0.0061 ms to 0.0295 ms per commit* in our study. To locate the corresponding method m, we leveraged GitHub's commit history, which conveniently displays the block of code change c along with the file path fp and the class cls or method m it belongs to. By collecting the method m while crawling the commit history, we can efficiently and accurately locate the affected GUI widgets W for RTS.

RQ3: Performing impact analysis using dynamic analysis can be as expensive as running one round of all test sequences as it requires code execution. The cost of creating initial test sequences for 20 apps ranged from 32 to 150 minutes, with 48.1% commit intervals falling within 30 minutes and 61.7% within 120 minutes. These results suggest that RTS is necessary to reduce the cost of regenerating all test sequences for each change. Additionally, the costs associated with instrumenting the code and generating/updating mapping information are relatively insignificant compared to the cost of executing the code. With the mapping information obtained from the initial analysis, related sequences can be quickly retrieved in just 0.03 ms after a code change.

2) RQ4: What is the cost of exercising the sequences containing affected UI and how does this cost relate to the time intervals between commits?: Costs of the RTS involves both the number of tests being selected and the time spent on executing these tests.

Number. Fig.3 shows the percentage of selected sequences. The reduction ranges from 0% to 92.7%, with an average is 67.06% We found the reduction rate was not as low as what we expected. We have identified two reasons. The first reason is because of the granularity of our dynamic analysis. Our approach tracks the changes at the method level. If a change occurs in a method, even if it is not directly associated with the UI, our approach will mark the UI as affected. A possible improvement is to perform a finer-granularity analysis, but with additional costs. Another reason is because of the presence of high-dependence UIs with numerous options. For instance, given the following sequences:
Sequence 1: $Home \rightarrow MoreOptions$
Sequence 2: $Home \rightarrow MoreOptions \rightarrow A$
Sequence 3: $Home \rightarrow MoreOptions \rightarrow B$
Sequence 4: $Home \rightarrow MoreOptions \rightarrow A \rightarrow A1$
Sequence 5: $Home \rightarrow MoreOptions \rightarrow B \rightarrow B1$
When the $MoreOptions$ is affected, all the above sequences would be selected for testing. However, the three sequences, $Home \rightarrow MoreOptions$, $Home \rightarrow MoreOptions \rightarrow A$, and $Home \rightarrow MoreOptions \rightarrow B$, are usually sufficient to test the functionalities of $MoreOptions$. One possible solution is to employ existing approaches, such as DetReduce [36] for minimizing Android GUI test suites in RTS for efficient testing selection.

Timing. The red curve in Fig.3 shows the average execution time of selected sequences. A 0-minute execution time implies either no impact on UI elements (e.g., configuration, documentation) or incomplete test coverage due to limited time or limitations in the test generation algorithm's ability to map code to UI elements. The minimum and maximum execution times are 5.25 and 34.3 minutes, respectively, indicating that the test case selection technique is effective at controlling the time interval to keep it within the desired interval (48.1% commit intervals falling within 30 minutes and 61.7% within

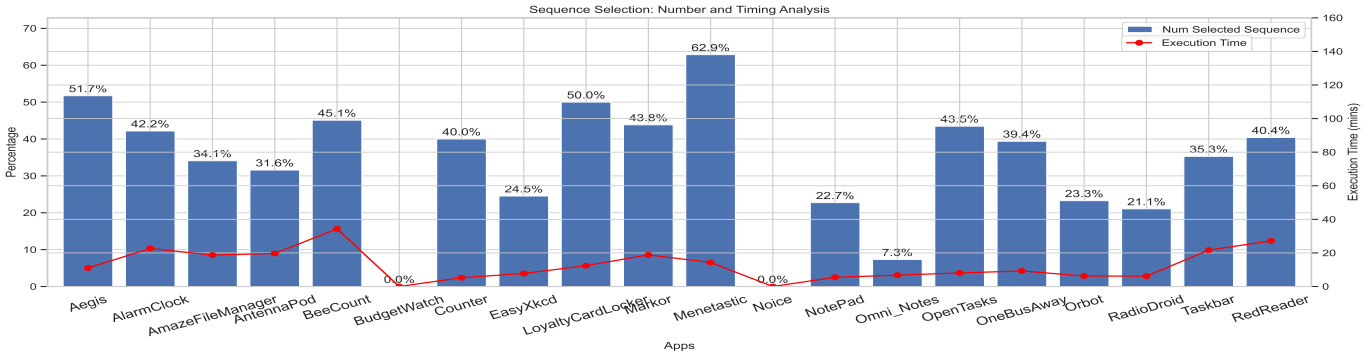

Fig. 3: Average Percentage of Selected Sequences

120 minutes.) and providing fast feedback to developers.

> **RQ4:** Even with straightforward selection techniques, the cost of testing can be significantly reduced by exercising selected sequences containing affected UI elements. This finding suggests that there is substantial potential for cost reduction if a repository of test sequences with high coverage, precise impact analysis, and optimal selection methods are developed and applied in future research.

C. Test Sequence Reuse Analysis

1) RQ5: What test executable rate of executing existing test sequences from the initial commit version to subsequent commit versions? To what extent existing test sequences can be reused across changes?: Test executable rate (TR) measures the percentage of test sequences that can be successfully executed in the current version (TS) compared to the total number of sequences generated from the initial version (TT): $TR = TS/TT$.

The ability to reuse existing test sequences across changes depends on various factors such as UI alterations, built version updates, and code refactoring. To evaluate the test executable rate, we executed all initial test sequences on subsequent commit versions. The change in the rate reflects the number of successful tests in the current version, providing insights into the degree of test sequence reuse for subsequent app versions. A higher test executable rate indicates better test sequence reuse and potential cost reduction.

Fig.4 displays the change of executable rate of running all existing test sequences as the commit is made and the application is updated. The analysis of the executable rate of existing test sequences among the 1475 commits shows interesting patterns in terms of re-test rates and trends. Among all subjects, it shows an overall downward trend, as expected. However, we have made several interesting observations:

Varying test executable rate. Among different apps, their test executable rates vary, ranging from 100% to 0%. Most apps keep relatively high re-test rates despite an overall downward trend. Some apps eventually drop to a 0% rate (e.g., EasyXkcd, BeeCount) while some maintained a 100% re-test rate (e.g., Noice, BudgetWatch).

Decreasing test executable rate. The test executable rate can decrease to various extents. In some cases, the rate may drop by only 2% at one time, while in others, it may experience a significant decrease of more than 8%. A minor decrease usually occurs when code changes cause only one or a few sequences to fail, often resulting from modifications to a leaf element or an element with limited dependencies in the UI hierarchy. In contrast, a larger decrease in the rate can arise from changes to a higher-level element or an element with a more complex set of dependencies within the UI hierarchy. Such changes have a broader impact, disrupting multiple navigation paths and causing a larger number of test sequences to fail to be executed. Assume that an app includes, but is not limited to, the following sequences: $Home{\rightarrow}Settings{\rightarrow}A$, $Home{\rightarrow}Settings{\rightarrow}B$, $Home{\rightarrow}Settings{\rightarrow}C$, and $Home{\rightarrow}Settings{\rightarrow}D$. If A is deleted, only one sequence might be affected, resulting in a minor decrease in the test executable rate. However, if the $Settings$ button is deleted, all functionality within $Settings$ would be impacted, causing a large number of sequences to fail. Consequently, the Test Executable Rate would experience a more significant drop due to the broader impact of the change.

Increasing test executable rate. There are two scenarios leading to the increase: 1) the addition of a previously deleted UI that exists in the original test cases, causing a revival in the reuse rate; and 2) the trend is determined by comparing the reuse rate of the current commit with that of the previous commit, in chronological order. However, the previous commit might not be the parent of the current commit. In such cases, the parent commit could be an earlier commit with a higher reuse rate, which can happen when multiple developers are working on the project and one makes several changes that cause the rate to drop, while another developer pushes a new commit based on an earlier commit. In both cases, the increase in reuse rate is observed when comparing the current commit to its parent commit.

Stable test executable rate. Our analysis revealed that a vast majority of commit versions, i.e., 98.68%, have the same executable as their previous version, indicating periods of stabilization throughout the testing process. As shown in Fig.4, these periods are represented by horizontal segments. The reason for such stability lies in the nature of the commits. Our study was based on commit versions rather than release ver-

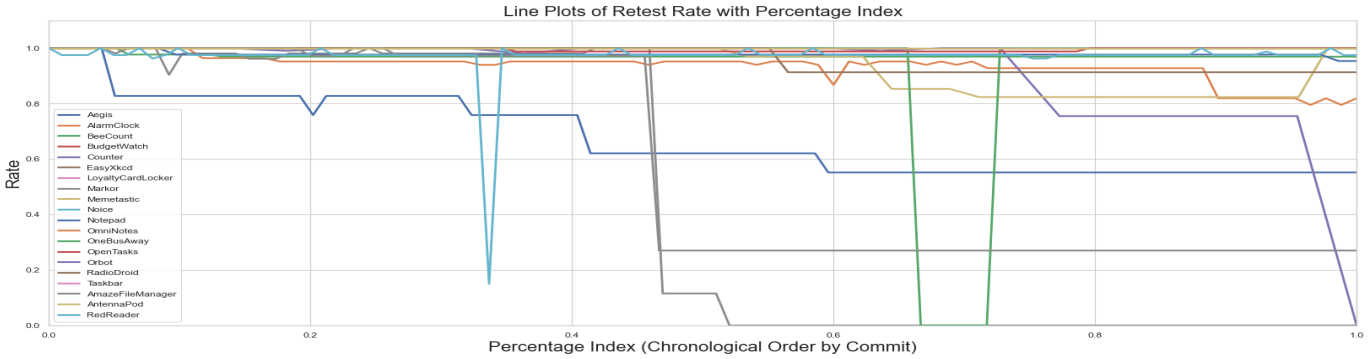

Fig. 4: Executable Rate

sions, which are typically achieved by merging multiple single commits from various developers. Developers usually make small, incremental changes to the code in each commit, and it is less common for them to make significant modifications, such as adding or deleting UI elements, in every commit. Such significant changes can have a substantial impact on the user experience, causing confusion or inconsistency to both users and other developers. As a result, consecutive commits often have the same executable, indicating stabilization periods. When a dramatic decrease in the test executable rate occurs, it is likely that the rate of subsequent commits will remain at the new level for a while until the next significant change is made.

RQ5: Test sequences that already exist are often highly reusable, meaning that they can be used multiple times. In order to maintain and improve this reusability, developers should follow good coding practices. This includes using unique element names, avoiding making changes or duplicating elements during development, and using distinctive UI element features to identify targets during automated testing. It's worth noting that the rate of reusable test sequences may fluctuate and even drop to 0% for some apps. However, periods of stabilization show that the rate of subsequent commits stabilizes at the new level before the next change. This indicates that by adjusting failed test cases or replacing them with new ones, developers can restore and maintain high reusability rates.

2) RQ6: What is the percentage of affected events that are commonly seen across changes? Are they focused on a small number of UI element?: In RQ4, we learned that UI elements that have multiple dependencies play an important role in the application and they should be prioritized in testing as well. In addition to the above prioritization, this RQ allows us to identify the percentage of affected UI elements that are commonly observed across various changes in applications, as well as to determine whether they are concentrated on a small number of events. If certain UI elements are more frequently changed over others, one can assume that these events need to be tested without analysis after a period of commits to avoid costs.

The percentage of a UI element that is commonly seen across changes is the ratio of the number of times the UI element is seen across changes (N) to the total number of occurrences of all UI elements that are commonly seen across changes (T): $P = N/T$. Each box in Fig.5 represents an individual app. The percentage values on the x-axis indicate the distribution of the proportion of a specific UI element frequently observed across changes for each app. A box with a higher median percentage value (the red line within each box) signifies that the app has UI elements that appear more frequently across changes. Conversely, a box with a lower median percentage implies that there is no frequent element being seen.

The results indicate that the median percentage of affected UI elements across changes ranges from 0% to 50%. There are few apps having a larger median percentage, around 30% and 50%, of affected UI elements while few have 0 % of affected UI elements. This implies that the extent of seen UI element depends on the specific app and its unique characteristics. The higher the median percentage of affected that are commonly seen across changes, the more likely it is that frequently testing them without extensive analysis is preferred. The fact that a few apps (e.g., Noice) have no UI element changes is because the continuous commits selected for studying involve only configuration and build updates, with no functional code changes. This usually happens when an app reaches maturity, developers tend to shift their focus to maintenance tasks. In such cases, developers might just need to focus their testing on core functionality and elements that have had defects in the past.

RQ6: There are certain UI elements commonly observed across changes in some applications, and they are not evenly distributed across all apps. Therefore, prioritizing testing of these UI elements without extensive analysis could be more efficient for certain apps, especially those with a higher percentage of affected UI elements that are commonly seen across changes. However, it is important to note that each app is unique, and the extent of seen UI element varies. Therefore, developers should consider the specific characteristics of their application before deciding on testing strategies.

V. Implications for Future Research

Our study motivates further research on RTS for mobile apps in CI environments.

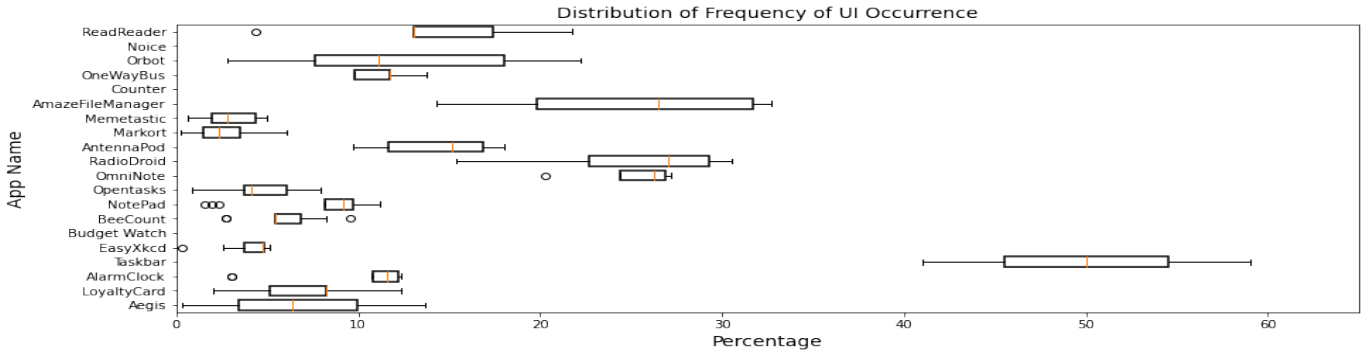

Fig. 5: Distribution of affected UI that are commonly seen across changes

A. Implications to Practitioners

Our study suggests that a significant proportion of open-source Android apps utilize Continuous Integration (CI) frameworks. While CI testing is commonly used to execute unit tests, our findings indicate that practitioners should pay attention to the selection and execution of UI tests to handle rapid code changes effectively.

Continuous Integration (CI) automates compilation, building, and testing to ensure code quality with every commit. For regression testing to work in CI, it must keep pace with frequent commits. Our Frequency Analysis reveals that 44.7% of functional changes occur within 30 minutes, and 52.9% occur within 60 minutes. These findings can assist developers in making testing decisions during development. For instance, if executing all test cases takes longer than 30 or 60 minutes, it would be advisable to apply Regression Test Selection (RTS) techniques selectively.

Our Test Reuse Analysis shows that existing test sequences often exhibit a high reuse rate since apps typically do not undergo significant changes over every commit. Moreover, alterations in the functional logic underlying the user interface typically do not affect the execution of test sequences. To maintain test sequences efficiently and ensure high reusability, developers must follow a consistent coding style, including avoiding the modification and duplication of GUI element features unless necessary, avoiding the use of common features, and eschewing the use of coordinates.

Thus, adaptive RTS techniques may be developed to selectively execute RTS for certain commits. For example, if a new project feature is introduced where more code commits are expected, RTS can be applied to reduce the cost of testing. Otherwise, it would be safer to execute more test cases.

B. Implications to Researchers

Test Automation. Our study highlights the importance of UI testing in mobile app development and the limitations of current CI testing practices that mainly focus on unit testing [1], [37], [38]. Although recent research has developed techniques to select events affected by code changes [5], as our results suggest, they cannot be applied in CI environments because the selected events have to be manually executed. We recommend the development of new RTS techniques that can automatically analyze impact events and execute associated event sequences to handle rapid code changes in CI environments. Furthermore, we suggest exploring the use of cost-efficient method analysis to facilitate the selection and execution of impacted events in UI testing.

Test Sequence Reuse. In Test Reuse Analysis, although existing sequences has high reusability across changes, they fail to cover newly introduced GUI elements and some UI logic changes. Regenerating test sequences using existing state-of-the-art tools can be a time-consuming process, as it requires running the tool for the entire application. This presents a significant motivation for developers to explore alternative methods for updating test sequences more efficiently.

By leveraging existing test sequences and incorporating impact analysis of code changes, developers can strive to generate new sequences and update existing sequences specifically targeting the updated GUI elements in RTS. This targeted approach ensures thorough testing of new features and updating the failed sequences without regenerating test sequences for the entire application, ultimately saving time and resources and enhancing RTS for UI testing. Additionally, the period of stabilization of test reuse observed in RQ5 further demonstrates that the new sequences generated through this targeted approach are not just one-time-use and likely to be reused in subsequent commit versions.

VI. Conclusions and Future Work

We have performed an empirical study employing regression test selection (RTS) in continuous integration (CI) environments. The study provides guidance for future research on RTS in CI environments. Future research can concentrate on developing advanced RTS techniques tailored to CI environments for mobile apps. This includes optimizing test cases selection algorithms, minimizing test cases that serve the same testing purpose, and generating new test cases or fixing failed ones based on existing sequences and code changes. By enhancing RTS techniques for UI testing in CI settings, developers can achieve more efficient and cost-effective software project delivery. As part of our future work, we will: (1) integrate the F-Droid repository, (2) determine the number and generation of test sequences for new functionalities across commits, (3) include static analysis and compare it with dynamic analysis, and (4) utilize state-of-the-art tools for test sequence generation.

Acknowledgments

This work was supported in part by U.S. National Science Foundation (NSF) under grants CCF-2246186, CCF-2140524, CCF-2152340, CCF-1909085, and CCF-1909763.

REFERENCES

[1] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 426–437.

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *STVR*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[3] T. Yu and T. Wang, "A study of regression test selection in continuous integration environments," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 135–143.

[4] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*, 2013, pp. 250–265.

[5] A. Sharma and R. Nasre, "Qadroid: regression event selection for android applications," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 66–77.

[6] Q. Do, G. Yang, M. Che, D. Hui, and J. Ridgeway, "Regression test selection for android applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 27–28.

[7] C. Peng, A. Rajan, and T. Cai, "Cat: Change-focused android gui testing," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 460–470.

[8] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: a large-scale empirical study," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 60–71.

[9] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 583–594.

[10] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 713–716.

[11] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *27th IEEE International Conference on Software Maintenance*, 2011, pp. 23–32.

[12] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *12th Asia-Pacific Software Engineering Conference*, 2005, pp. 9–pp.

[13] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *JOOP*, vol. 8, no. 2, pp. 51–65, 1995.

[14] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 46–53.

[15] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 111–122.

[16] Y. Li, Z. Yang, Y. Guo, and X. Chen, "A deep learning based approach to automated android app testing," *arXiv e-prints*, pp. arXiv–1901, 2019.

[17] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[18] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, vol. 48, no. 10, 2013, pp. 641–660.

[19] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.

[20] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[21] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[22] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[23] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 596–607.

[24] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[25] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[26] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, 2016.

[27] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP),*, 2017, pp. 233–242.

[28] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 805–816.

[29] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 128–139.

[30] "Scrapy," https://en.wikipedia.org/wiki/Scrapy.

[31] J. Tao, Q. Zhao, P. Cao, Z. Wang, and Y. Zhang, "Apk-dfs: An automatic interaction system based on depth-first-search for apk," in *Algorithms and Architectures for Parallel Processing: 17th International Conference, ICA3PP 2017, Helsinki, Finland, August 21-23, 2017, Proceedings 17*. Springer, 2017, pp. 420–430.

[32] "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey.html.

[33] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[34] "Dinodroid," https://github.com/softwareTesting123/DinoDroid.

[35] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[36] W. Choi, K. Sen, G. Necula, and W. Wang, "Detreduce: minimizing android gui test suites for regression testing," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 445–455.

[37] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing junit test cases in absence of coverage information," in *IEEE International Conference on Software Maintenance*, 2009, pp. 19–28.

[38] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, 2012.