Amplifying Tests for Cross-Platform Apps through Test Patterns

Thiago Botti de Assis, André Augusto Menegassi, and Andre Takeshi Endo
Department of Computing
Federal University of Technology – Parana (UTFPR)
Cornélio Procópio, Brazil
thiagobotti@gmail.com, andremenegassi@hotmail.com, andreendo@utfpr.edu.br

***Abstract*—Cross-platform app development has achieved meaningful results in practice with frameworks like React Native, Xamarin, and Apache Cordova. Unlike native apps, such frameworks support the development of a mobile app that can run in different platforms. Nevertheless, the literature lacks techniques to test cross-platform apps since most of the existing works focus on native Android apps. A promising strategy for native apps is to amplify test suites so that the specific characteristics of mobile apps can be tested. This paper aims to investigate the test amplification of cross-platform apps. To do so, we apply four test patterns that verify well-known characteristics of mobile computing and amplify existing test suites. The proposed approach has been implemented in a tool capable of generating Appium test scripts and was evaluated with nine cross-platform apps. The amplified test suites exercise new scenarios, uncovering 23 unique bugs in eight out of nine apps.**

I. Introduction

Smartphones, tablets, e-readers, and wearables have dominated the consumption of digital information and data traffic; the number of these devices has grown exponentially over the years [1]. The software running in such devices, so-called *mobile apps* (or just *apps* for short), radically changed the lifestyle of billions of people around the world, being used for several hours every day and helping users to perform a wide variety of activities [2]. To meet the ample range of users, millions of apps have been developed and are available for download [3]. Currently, the dominant operating systems (OSs) for mobile devices are Google's Android and Apple's iOS; they provide their own software development kits, characterizing the single-platform native development.

In order to reach more end users, the software industry has shown an upward trend in adopting cross-platform mobile app development. In such a paradigm, a unique code base is used to deliver the same app in different platforms (e.g., Android, iOS, Windows) and their versions [4], [5]. This is aided by several open source and commercial cross-platform development frameworks and tools; the most notable examples are React Native, Xamarin, and Apache Cordova. Developed by Facebook, React Native leverages the well-known web library React to generate mobile apps with native user interfaces (UIs) and coded in Javascript [6]. Acquired by Microsoft in 2016, Xamarin offers C# or F# as a common programming language to develop apps for the three main platforms: Android, iOS and Windows. According to Microsoft, 75% of the code is shared among the platforms on average; this figure can be close to 100% if the UI toolkit Xamarin.Forms is employed [7]. Finally, Apache Cordova fosters the development of hybrid apps using standard web technologies like HTML5, CSS3, and Javascript [8]. As such, it is popular among web developers and has supported other mobile frameworks like Ionic and PhoneGap [9], [10].

Apart from this trend, research efforts in engineering high-quality mobile apps have focused on native development for Android [11], [12]. The demand for quality in mobile apps has grown along with their spread; the users want the apps to be reliable, robust, and efficient. As a consequence, mobile software engineers must adopt adequate quality assurance techniques [2]. Software testing is the primary activity to reduce risks and minimize the presence of bugs in the software. In a nutshell, testing involves the process of running a software system with the purpose of uncovering bugs [13].

As mobile apps possess specific features to be checked, *Test Amplification* [14] has been investigated in literature to go beyond functional test cases. Test amplification can be defined as the extension of existing test cases to verify other properties of the software under test. This is a promising strategy for mobile apps that are susceptible to bugs related to device rotation, interruptions, loss of resources, unique UI events/components, and so on. The verification of such specific features can be described in a well-defined set of generic actions and expected behavior called *test pattern* [15], [19]. The amplification by test patterns has been explored in native Android apps and results have shown effective bug detection [15], [16], [18], [19]. However, little is known on how test amplification performs in mobile apps developed with cross-platform frameworks.

This paper introduces an approach for test amplification of cross-platform apps. In particular, we describe four test patterns that aim to verify well-known characteristics of mobile computing and amplify test cases. The proposed approach has been implemented in a tool, called x-PATeSCO, capable of producing test scripts for cross-platform apps. We evaluated the approach and its supporting tool with nine real-world apps, developed with frameworks React Native, Xamarin, and Cordova.

This paper is organized as follows: Section II presents the

DOI reference number: 10.18293/SEKE2019-076

background and related work. Section III brings the test amplification approach proposed for cross-platform apps. Section IV presents the evaluation setup and the results are analyzed in Section V. Finally, Section VI makes the concluding remarks and sketches future work.

II. Background and Related Work

Mobile computing brings a diverse set of features that may cause failures in the software developed for such platform. Holl and Elberzhager [22] propose a failure classification based on a literature review and a study with developers and testers of mobile apps. They list events that might make apps to fail, namely, temporarily disconnected network, network change (3G - WiFi), GPS temporarily disconnected, weak battery, globalization (localization, language, date or time differences between clients and servers), UI changes due to orientation, screen changes, OS-specific native functionality, interruptions via incoming calls or text messages, unexpected termination of an app process, and permissions. Similarly, Linares-Vásquez et al. [21] manually searched for bugs in Android apps related to specific features like GPS failures, loss of access to network data, slow data reception, and device rotation. The authors used this bug set to elaborate Android-specific mutation operators and developed a mutation testing tool for Android apps.

These well-known features have also been used to amplify existing test cases in a systematic way. Zaeem et al. [16] also performed a study with bugs from the most popular open source app projects, identifying feature interactions that might cause errors. The authors proposed an approach to verify automatically the feature interactions and a tool was developed to exhaustively test them. Morgado and Paiva [15] introduce a catalog of behavioral principles of the Android UI elements. Such behavioral principles are described as test patterns related to UI menu elements (drawer style) and device rotation. They also propose patterns related to features like GPS and 3G connection. Adamsen et al. [18] explore the injection of adverse conditions in existing test suites. The adverse conditions are related to common usage of apps like pause-stop-resume, rotation, as well as missing or changed resources such as loss of GPS accuracy, change of mobile network, and transition between Internet networks (3G-WiFi-3G). Their results show that the approach is effective in finding critical flaws and the additional cost in the test time is low, compared to the number of bugs found.

While test amplification and test patterns have been investigated [15], [16], [18], the state of the art in mobile app testing is focused on native Android apps [11], [12]. On the other hand, cross-platform app development has gained momentum by winning a meaningful market share, and being supported by big players of the software industry like Facebook, Microsoft, and Apache Software Foundation. In this context, one might wonder if the testing techniques developed for native apps are seamlessly applicable to cross-platform apps. In this paper, we inquire into test amplification for cross-platform apps; to our best knowledge, this is the first study on the topic.

III. Test Amplification Approach

This section describes the approach we define to test amplification of cross-platform apps; Figure 1 shows an overview of the approach. There is a component called `Test Amplification Generator` that receives as input a set of system-level test cases $TC_1, TC_2, ..., TC_n$ and a one or more test patterns. Then, it produces as output a combination of test cases and test patterns, the *amplified test script*. Such scripts can then be executed in the *app under test* (AUT).

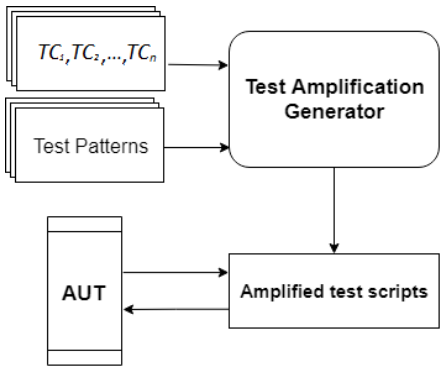

Fig. 1. Overview of the proposed approach.

The following sections explain how we implemented the approach. Section III-A details how a test case and a test pattern are combined to amplify the tests. In particular, we leverage four test patterns that verify well-known characteristics of mobile apps. Section III-B focuses on the `Test Amplification Generator`, describing how we made the approach applicable to cross-platform apps.

A. Test Patterns

We herein describe the four test patterns adopted in this work. All test patterns assume that a test case is provided as a sequence of system-level events, mostly UI actions like clicking a button, selecting an item, or filling a text field. The test pattern contains not only a pre-defined sequence of specific events that should be executed, but also a mechanism to identify a potential bug (i.e., a test oracle). To simplify the presented algorithms, we do not treat exceptions that can be thrown when a given event is executed in the app (e.g., a crash). Nevertheless, our tool deals with it by capturing the exception and recording the bug information in a log (further details in Section III-B). The literature on Android app testing reports different strategies to amplify test suites [4], [16], [18], [21]; this study adopts four test patterns, well-known in literature and that we found relevant to cross-platform apps.

1) Lost Connection (LC): This pattern aims to verify how the AUT behaves when the Internet connection is lost [15], [21], [22]. As mobile devices are susceptible to unstable networks, the AUT is supposed to be aware and react when a connection is lost. By reacting, we assume that some feedback is provided to the end user, like an error message or a warning [17], [20].

Algorithm 1 describes how a test case is amplified to verify a lost connection. It receives as input a test case as an array of events, and a map of states recorded during the original execution of the test case. First, the procedure turns the connection off and starts the app (Lines 4-5). In each iteration, it checks if the current state is different from the original state (Line 8); i.e., the connection lost had an impact on the UI. If the current state has no sign of feedback for the user (like an error message) (Line 9), we record the scenario as a bug (Line 10). To resume the verification in the following events of the test case, we turn the connection on (Line 12), execute the events normally until the current state (Line 13). The connection is then turned off (Line 14) and the iteration continues to the next event (Line 16).

Algorithm 1 Lost Connection Test Pattern

```
1: procedure LostConnection(allEvents[], origStates[])

2:     input allEvents[] - Test case as an array of events
3:     input origStates[] - Map of states collected from the original execution

4:     connection( OFF );
5:     startApp();
6:     for each event ∈ allEvents do
7:         currentState = getActualState();
8:         if currentState != origStates[event] then //changed GUI
9:             if ! existLostConnectionMsg(currentState) then //no feedback
10:                RecordBugInfo();
11:            end if
12:            connection( ON );
13:            bringAppTo( currentState );
14:            connection( OFF );
15:        end if
16:        executeEvent(event);
17:    end for
18: end procedure
```

2) Back Event (BE): This pattern aims to verify how the AUT behaves when the Back event is triggered during a test case [16], [22]. We assume that when running, the Back event must go to a previous screen of the application [17], [20].

Algorithm 2 describes how the test case is amplified to verify the execution of the Back event. It receives as input a test case as an array of events. Initially, the procedure starts the application (Line 3). In each iteration, we execute the corresponding event, then it is checked if the state after the event execution is equal to the state before execution (Line 8). The procedure executes the Back event, and validates whether the application has returned to the previous state (Lines 9-11). If the states are different after running the Back event, we record this scenario as a bug (Line 12). If the states are equal, the procedure follows the verification for the next event.

3) Side Drawer Menu (SDM): This pattern aims to verify how AUT behaves when having a side menu [16], [22]. The menu opens when the user slides from the screen to the center or clicks in a menu icon or label [15]. We validate if all the menu items take the app to a different screen.

Algorithm 3 describes how the test case is amplified to validate the application menus. It receives as input a test case as an array of events. Initially, the procedure starts the application (Line 3). For each iteration, we check whether the application has a menu (Lines 6-23). If there is a menu in

Algorithm 2 Back Event Test Pattern

```
1: procedure BackEvent(allEvents[])

2:     input allEvents[] - Test case as an array of events

3:     startApp();
4:     for each event ∈ allEvents do
5:         beforeState = getActualState();
6:         executeEvent(event);
7:         afterState = getActualState();
8:         if beforeState != afterState then //there is a screen transition
9:             ExecuteBack();
10:            afterBackState = getActualState();
11:            if beforeState != afterBackState then //bug detected
12:                RecordBugInfo();
13:            end if
14:            bringAppTo( afterState );
15:        end if
16:    end for
17: end procedure
```

the AUT, the procedure checks the menu items empirically by clicking on them and checking if the application changes the screen after entering a menu item (Lines 28-33). If the application does not change screen, we record this scenario as a bug (Line 31). If no menu item is found, the procedure proceeds to the next iteration (Line 35).

Algorithm 3 Side Drawer Menu Test Pattern

```
1: procedure SideDrawerMenu(allEvents[])

2:     input allEvents[] - Test case as an array of events

3:     startApp();
4:     for each event ∈ allEvents do
5:         beforeMenuState = getActualState();
6:         haveMenu = false;
7:
8:         //Try do detect menu by horizontal swipe
9:         initialState = beforeMenuState;
10:        Swipe();
11:        finalState = getActualState();
12:        if initialState != finalState then
13:            haveMenu = true;
14:        else
15:            //Try do detect menu by menu button
16:            initialState = getActualState();
17:            LocateAndClickMenuButton();
18:            finalState = getActualState();
19:
20:            if initialState != finalState then
21:                haveMenu = true;
22:            end if
23:        end if
24:
25:        if haveMenu then
26:            menuState = getActualState();
27:            for each item ∈ getMenuItems() do
28:                tap( item );
29:                afterTapItemState = getActualState();
30:                if afterTapItemState ∈ {beforeMenuState, menuState} then
31:                    RecordBugInfo();
32:                end if
33:                bringAppTo(menuState);
34:            end for
35:            bringAppTo(beforeMenuState);
36:        end if
37:        executeEvent(event);
38:    end for
39: end procedure
```

4) Don't Change State (DCS): This pattern aims to verify how the AUT behaves when external events triggered by the user occur during the test case [15], [16], [18]. As mobile devices are susceptible to these actions, the AUT is supposed

to be aware and prevent failures because of external events. We assume that after the execution of external events, the app should return to the state it was before [17], [20].

Algorithm 4 describes how a test case is amplified to verify the external events performed by users. It receives as input a test case as an array of events, and the type of the external action to be executed. The types currently supported are Rotate, Pause-Stop-Resume, Zoom, Swipe, and Scroll. First, the procedure starts the application (Line 4). For each iteration, it records the state before and after the external event (Lines 7-9). If the states are different (the AUT did not return to the same screen) (Line 10), we record the scenario as a bug (Line 11). The procedure then continues to the next event (Line 12).

Algorithm 4 Don't Change State Test Pattern

```
1: procedure DONTCHANGESTATE(allEvents[], type)
2:     input allEvents[] - Test case as an array of events
3:     input type ∈ {Rotate, PauseStopResume, Zoom, Scroll, Swipe}
4:     startApp();
5:     for each event ∈ allEvents do
6:         executeEvent(event);
7:         beforeState = getActualState();
8:         execute( type ); //execute one of the 5 types
9:         afterState = getActualState();
10:        if beforeState != afterState then //didn't come back to the same screen
11:            RecordBugInfo();
12:            bringAppTo( beforeState );
13:        end if
14:    end for
15: end procedure
```

B. Test Amplification Generator

We developed a tool named x-PATeSCO (*cross-Platform App Test Script reCOrder*) to support cross-platform mobile app testing. The tool is built on top of Appium [23], an open source framework to automate UI tests in native, Web and hybrid apps. Moreover, Appium is cross-platform and makes it possible to automate tests for iOS and Android platforms, using a Selenium WebDriver API. x-PATeSCO is able to read the elements of an app's UI and assist the tester in selecting these elements to design automated test cases.

The `Test Amplification Generator` component has been implemented within x-PATeSCO. The four test patterns aforementioned are embedded into the tool as code templates, that are integrated with the recorded test cases. Such integration is realized when the test scripts are generated; currently, x-PATeSCO produces scripts coded in C#. These scripts are capable of executing the four test patterns in the original test case automatically, therefore no manual effort is required to apply test amplification in cross-platform apps.

IV. Evaluation Setup

To evaluate the proposed approach, we set out the following research questions (RQs):

- **RQ**$_1$. Can test amplification detect bugs in cross-platform apps?
- **RQ**$_2$. What is the impact of test amplification on test execution?
- **RQ**$_3$. Are the results replicable in different settings (e.g., OS version)?

To answer **RQ**$_1$, we record the number of unique bugs detected as a consequence of the test amplification. As our approach may show false alarms, the number of tests presenting failure (failures) and false positives are also collected. So, for each failure, we try to reproduce it by manually performing the test steps. For a successful reproduction, we have a bug, otherwise a false positive. As for **RQ**$_2$, we measure the overhead of the amplified tests in test execution time. Given that T_{orig} is the runtime of the original test cases and T_{amp} is the runtime of the amplified tests, the overhead is calculated by T_{amp}/T_{orig}. Finally, **RQ**$_3$ analyzes the results collected in RQ$_1$ from the perspective of different configurations. In this study, we only considered different versions of the OS Android.

Table I lists the nine apps evaluated; for each app, it shows the project size in number of lines of code (LOC), the cross-platform app development framework, the number of test cases (#TCs) and their events (#Ev.). Such cross-platform apps' size ranges from around 400 LOC to up to 178 KLOC, and are implemented in different cross-platform development frameworks (namely, Apache Cordova, React Native, and Xamarin). Two apps are industrial (namely, `MemesPlay` and `Bargains`) and have been provided by partner IT companies. The other seven are open source projects obtained from GitHub. Each app has two or three test cases (with nine to 20 events each), designed by independent participants.

TABLE I
Apps under test.

App	Size-LOC	Framework	#TCs (#Ev.)
`Fresh-Food-Finder`	13824	Cordova	3 (20)
`OrderApp`	71565	Cordova	3 (16)
`MemesPlay`	5484	Cordova	3 (12)
`Agenda`	1038	Cordova	3 (18)
`TodoListCordova`	9304	Cordova	3 (10)
`MovieApp`	2088	React Native	3 (10)
`TodoList`	405	React Native	3 (13)
`Tasky`	654	Xamarin	3 (9)
`Bargains`	178266	Xamarin	2 (10)

x-PATeSCO was used to support the experimental evaluation; it connects to Appium which, in turn, connects to mobile devices to run the test cases and log pieces of information to answer **RQ**$_1$, **RQ**$_2$, and **RQ**$_3$. The apps were installed in two devices, one with Android 6.0 and another with Android 7.0, and all tests were executed in both devices. Besides the mobile devices, the test projects were run from a computer with an Intel Core i5 dual-core (2.5 GHz) processor and 8 GB of RAM without any further processing or communication load in order to avoid CPU and memory saturation. An Appium server was also installed in this machine.

To foster future replication and overcome possible threats to this study, we make the tool and the experiment objects available as an open source experimental package in `https://goo.gl/7CF4oZ`

V. Analysis of Results

a) Bug detection: Table II shows the number of bugs found, failing test cases (Fail), and false positives (FP); rows identify the apps and OS version (OS V.), while columns group the test patterns: *Back Event* (*BE*), *Lost Connection* (*LC*), *Side Drawer Menu* (*SDM*), and *Don't Change State* (*DCS*). Column *DCS* takes into account the results of Rotate, PauseStopResume, Zoom, Scroll, and Swipe. Symbol '-' indicates that the test pattern is not applicable in a given app. For instance, `Agenda` does not use Internet connection, so *LC* is not applicable. In this part, we focus our analysis on the OS version that had more bugs revealed (namely, Android 6.0).

As for *BE*, this pattern revealed seven bugs in seven different apps. We did not observe false positives since all failures were caused by the identified bugs. In general, this pattern finds a faulty scenario in which the app does not return to a previous screen using the back button. Instead, the app terminates, goes to the background, or returns to a different screen. Concerning *LC*, this pattern revealed five bugs in four out of five apps in which is applicable. It shows a smaller number of failures and no false positives. This is likely due to the low number of events that depend on an Internet connection to properly execute. For instance, `OrderApp` does not give a feedback on a lost message due to a failing Internet connection. `MovieApp` presents the same bug, and in other scenario of connection lost the app crashes.

There is no bug, failure or false positive for *SDM*. While four apps have menus, all menu items are accessible and navigate to different screens correctly. We surmise that menus are built with well-tested UI components and developers pay special attention to them due to their relevancy in the app. As a consequence, menus are more tested and less susceptible to bugs. The *DCS* test pattern uncovered 11 bugs in eight apps. Notice that this pattern showed a high number of failures and false positives. This occurred due to issues related to the Appium framework. Actions like Zoom, Scroll and Swipe caused exceptions during the automated test execution, but we failed to reproduce these scenarios manually. As for the bugs, the `OrderApp`, `TodoList`, `Bargains`, and `Memesplay` apps restarted after a Pause-Stop-Resume action. `Agenda`, `Fresh-Food-Finder`, `MovieApp`, `TodoList`, and `Bargains` slowed down when performing the zoom action.

The amplified tests detected 23 unique bugs in eight out of the nine cross-platform apps. Among the four test patterns, only *SDM* did not reveal a bug. A reasonable number of false positives has been observed, except for *DCS*. We believe that new Appium versions and adjustments in x-PATeSCO can reduce the presence of false alarms in future. The results give evidence that supports a positive answer for $\mathbf{RQ}_1$.

b) Overhead in test execution: Table III shows the overhead of the amplified tests in test execution time. The overhead of the test patterns shows some variation among the apps. For instance, the *BE*'s overhead ranges from 0.71 to 4.5 times in comparison with the original test suite. On average, *BE* also poses the highest overhead (2.97x), while *SDM* shows the lowest average (2.06x). *DCS* and *LC* had intermediate averages, 2.42x and 2.68x, respectively.

TABLE III
Overhead in the test execution time.

App	OS V.	BE	LC	SDM	DCS
`Fresh-Food-Finder`	6.0	4.50	1.70	1.71	2.59
	7.0	4.55	1.72	1.70	2.69
`OrderApp`	6.0	4.25	3.37	-	3.91
	7.0	3.36	2.66	-	3.09
`Memesplay`	6.0	4.25	5.56	3.09	4.47
	7.0	2.40	3.58	1.96	2.70
`Agenda`	6.0	3.44	-	1.82	2.07
	7.0	3.42	-	1.64	2.29
`TodoListCordova`	6.0	3.71	-	-	2.62
	7.0	4.17	-	-	2.87
`MovieApp`	6.0	2.20	2.54	3.10	2.04
	7.0	0.71	0.82	1.02	0.65
`TodoList`	6.0	1.43	-	-	2.40
	7.0	1.39	-	-	1.99
`Tasky`	6.0	1.97	-	-	2.38
	7.0	2.00	-	-	2.43
`Bargains`	6.0	2.22	3.04	-	1.77
	7.0	2.50	2.31	-	1.99
	Max	4.50	5.56	3.10	4.47
	Min	0.71	0.82	1.02	0.65
	Avg	2.97	2.68	2.06	2.42

The results show that amplified tests execute from 0.65 to up to 5.56 times the runtime of the original test suite. Answering $\mathbf{RQ}_2$, the impact can be reasonable depending on the app and test pattern, though the (low) cost of CPU time might be compensated by the bug detection capabilities of the proposed approach.

c) Different settings: Due to space limitations, we analyze the results using two settings: Android 6.0 and Android 7.0. As we used two devices with different hardware configurations, variations in the runtime were expected and natural (see Table III). So, we focus on the results summarized in Table II. Notice that there is no discrepancy between the OS versions for test patterns *BE*, *LC*, and *SDM*. So, the 12 bugs detected by them were detected in both devices. From the 23 bugs found in Android 6.0, 11 were not observed in Android 7.0. Fewer failures were also noticed in Android 7.0, yet the number of false positives was slightly higher.

As all false positives were caused by Appium's issues, newer versions of Android seem to be more robust for automated tests. Surprisingly, all bugs found by *DCS* in Android 6.0 were not detected in Android 7.0. We believe that Android policies for background processes (app not in focus) may differ between the OS versions. This might cause a faulty behavior in a specific version.

The different results obtained in Android 6.0 and Android 7.0 give initial evidence that supports a negative answer to $\mathbf{RQ}_3$. This motivates further investigation on different Android versions and other target OSs like iOS and Windows.

TABLE II
NUMBER OF BUGS, FAILURES (FAIL), AND FALSE POSITIVES (FP).

App	OS V.	BE			LC			SDM			DCS			Total		
		Bug	Fail	FP	Bug	Fail	FP	Bug	Fail	FP	Bug	Fail	FP	Bug	Fail	FP
Fresh-Food-Finder	6.0	1	10	0	0	0	0	0	0	0	1	31	15	2	41	15
	7.0	1	10	0	0	0	0	0	0	0	0	16	16	1	26	16
OrderApp	6.0	1	15	0	1	6	0	-	-	-	1	9	0	3	30	0
	7.0	1	15	0	1	6	0	-	-	-	0	9	9	2	30	9
Memesplay	6.0	1	8	0	1	6	0	0	0	0	1	23	21	3	37	21
	7.0	1	8	0	1	6	0	0	0	0	0	17	17	2	31	17
Agenda	6.0	1	8	0	-	-	-	0	0	0	1	40	38	2	48	38
	7.0	1	8	0	-	-	-	0	0	0	0	12	12	1	20	12
TodoListCordova	6.0	1	3	0	-	-	-	-	-	-	1	11	10	2	14	10
	7.0	1	3	0	-	-	-	-	-	-	0	11	11	1	14	11
MovieApp	6.0	1	8	0	2	6	0	0	0	0	3	46	39	6	60	39
	7.0	1	8	0	2	6	0	0	0	0	0	46	46	3	60	46
TodoList	6.0	0	0	0	-	-	-	-	-	-	1	18	13	1	18	13
	7.0	0	0	0	-	-	-	-	-	-	0	15	15	0	15	15
Tasky	6.0	0	0	0	-	-	-	-	-	-	0	2	2	0	2	2
	7.0	0	0	0	-	-	-	-	-	-	0	2	2	0	2	2
Bargains	6.0	1	3	0	1	5	0	-	-	-	2	21	5	4	29	5
	7.0	1	3	0	1	5	0	-	-	-	0	20	20	2	28	20
Total	6.0	7	55	0	5	23	0	0	0	0	11	201	136	23	279	143
	7.0	7	55	0	5	23	0	0	0	0	0	148	148	12	226	148

VI. CONCLUSION

This paper presented and evaluated an approach to generate amplified test scripts for cross-platform apps. To do so, we employed four test patterns that check well-known features of a mobile application: namely, *Lost Connection*, *Back Event*, *Side Drawer Menu*, and *Don't Change State*. We developed a supporting tool called x-PATeSCO and nine apps were used in the experiments. The results gave evidence that test amplification is capable of uncovering new bugs in cross-platform apps, though there exists a reasonable overhead in the test execution time. We also noticed variations between different versions of Android; this motivates further investigation on automated tests in multiple configurations of hardware, OSs, and so on.

ACKNOWLEDGMENT

Andre T. Endo is partially financially supported by CNPq/Brazil (grant number 420363/2018-1).

REFERENCES

[1] Dzhagaryan, A. and Milenkovi, A. (2016) "Models for Evaluating Effective Throughputs for File Transfers in Mobile Computing", In: ICCCN 2016. doi: 10.1109/ICCCN.2016.7568547.

[2] Amalfitano, D., Riccio, V., Paiva, A. C. R. and Fasolino, A. R. (2017) "Why does the orientation change mess up my Android application? From GUI failures to code faults", STVR journal (September), p. 1–27.

[3] Rumee, S. T. A. and Liu, D. (2013) "DroidTest: Testing Android Applications for Leakage of Private Information", In: ISC 2013.

[4] Joorabchi, M. E., Ali, M. and Mesbah, A. (2015) "Detecting inconsistencies in multi-platform mobile apps", In ISSRE 2015, p. 450–460. doi: 10.1109/ISSRE.2015.7381838.

[5] El-kassas, W. S., Abdullah, B. A., Yousef, A. H., Member, S. and Wahba, A. M. (2016) "Enhanced Code Conversion Approach for the Integrated Cross-Platform Mobile Development (ICPMD)", 42(11), p. 1036–1053.

[6] Build native mobile apps using JavaScript and React. Available at: https://facebook.github.io/react-native/ (Accessed: March 2019).

[7] Platform Xamarin. Available at: https://www.xamarin.com/platform (Accessed: March 2019).

[8] Apache Cordova. Available at: https://cordova.apache.org (Accessed: March 2019).

[9] Malavolta, I., Ruberto, S., Soru, T. and Terragni, V. (2015) "Hybrid Mobile Apps in the Google Play Store: An Exploratory Investigation", p. 56–59. doi: 10.1109/MobileSoft.2015.15.

[10] Bosnic, S., Papp, I. and Novak, S. (2016) "The development of hybrid mobile applications with Apache Cordova". doi: 10.1109/TELFOR.2016.7818919.

[11] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé and J. Klein, "Automated Testing of Android Apps: A Systematic Literature Review," 2018 in IEEE Transactions on Reliability. doi: 10.1109/TR.2018.2865733

[12] Zein, S., Salleh, N. and Grundy, J. (2016) "A systematic mapping study of mobile application testing techniques", Journal of Systems and Software. Elsevier Inc., 117, p. 334–356. doi: 10.1016/j.jss.2016.03.065.

[13] Myers, G. J., Thomas, T. M. and Wiley, J. (2004) The Art of Software Testing.

[14] Danglot, B., Vera Perez, O., Yu, Z., Monperrus, M., and Baudry, B. (2017) "A Snowballing Literature Study on Test Amplification". CoRR. Available at: https://arxiv.org/abs/1705.10692

[15] Morgado, I. C. and Paiva, A. (2015b) "The iMPAcT Tool: Testing UI Patterns on Mobile Applications", 30th IEEE/ACM International Conference on Automated Software Engineering The, p. 876–881. doi: 10.1109/ASE.2015.96.

[16] Zaeem, R. N., Prasad, M. R. and Khurshid, S. (2014) "Automated generation of oracles for testing user-interaction features of mobile apps", In: IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014, p. 183–192. doi: 10.1109/ICST.2014.31.

[17] Android Core App Quality. Available at: https://developer.android.com/docs/quality-guidelines/core-app-quality (Accessed: March 2019).

[18] Adamsen, C. Q., Mezzetti, G. and Møller, A. (2015) "Systematic Execution of Android Test Suites in Adverse Conditions", In: The International Symposium on Software Testing and Analysis (ISSTA).

[19] Morgado, I. C. and Paiva, A. (2015a) "Testing approach for mobile applications through reverse engineering of UI patterns", 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), p. 42–49. doi: 10.1109/ASEW.2015.11.

[20] Apple iOS Human Interface Guidelines. Available at: https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/ (Accessed: March 2019).

[21] Linares-vásquez, M., Bavota, G., Tufano, M., Moran, K., Penta, M. Di, Vendome, C., Bernal-cárdenas, C. and William, C. (2017) "Enabling Mutation Testing for Android Apps". doi: 10.1145/3106237.3106275.

[22] Holl, K. and Elberzhager, F. (2014) "A Mobile-specific Failure Classification and its Usage to Focus Quality Assurance". doi: 10.1109/SEAA.2014.19.

[23] Appium. Available at: http://appium.io/ (Accessed: March 2019).