Regression Event Selection for Android Applications

Anonymous Author(s)

ABSTRACT

Popular Android applications undergo frequent releases. The new app versions demand a substantial amount of effort from the testing team. Ensuring functional testing of the new features, as well as regression testing of the previous functionality, are time-consuming and error-prone. Literature is aplenty with indications of the dominant prevalence of *manual* testing for Android applications. Thus, there is a need for a tool that eases the testing efforts as well as saves the overall time of the product release cycle. In this work, we present QADroid, *the first activity- and event-aware regression selection tool for Android apps*. Salient features of QADroid are: (i) a richer change-set analyzer that covers code as well as non-code components for regression, (ii) it presents a pictorial representation of the app's functioning, and (iii) it displays the regression points in the app as a mapping between *activities* to *user-elements* to *events*. Features (ii) and (iii) help the testers in understanding the technical findings better. We evaluated QADroid on 1006 releases of 50 open source Android projects. The results show that QADroid reduced the activity selection by 58% and event selection by 74% compared to the traditional way of exhaustive testing of all activities and events, thereby significantly reducing the manual testing efforts.

ACM Reference Format:
Anonymous Author(s). 2019. Regression Event Selection for Android Applications. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mobiles have gained widespread acceptance for online computing, the place web applications used to enjoy sometime back [1, 27]. This has led to the invention of the buzzword *mobile-first* for companies that actively target their business model at mobile customers [16, 26]. Several companies (e.g., Uber [53], Amazon [4]) have started or relaunched their businesses on mobile platforms. At the core of several of these businesses lies Android [6], one of the most popular platforms for building mobile applications [46], which we focus on in our study. Android applications are updated frequently to ensure keeping up with the user needs and the business demands. Such updates can be as small as minor bug fixes, and as big as adding new functionality. Table 1 lists the number of updates per month for ten popular apps. As it can be seen many apps release an update almost every week. The updated app needs to be tested for the new changes and also to ensure that the original functionality continues to work as expected.

Table 1: Release frequency of ten popular apps [9]

App	#Versions	Since	Avg (per month)
Amazon	65	Dec 10, 2014	~2
Facebook	168	Apr 16, 2014	~3
Instagram	153	Apr 22, 2014	~3
Netflix	66	Apr 22, 2014	~1
Snapchat	193	Jan 17, 2014	~4
Truecaller	110	Apr 24, 2014	~2
Twitter	220	Apr 21, 2014	~4
Uber	194	Mar 13, 2014	~4
Whatsapp	70	Apr 27, 2014	~1
Youtube	75	May 20, 2016	~3

Regression testing [20, 31, 47–49] is a well-established software testing technique which ensures that the incremental updates or the enhancements to the software do not break the existing functionality. There is a multitude of regression testing solutions for desktop/web applications, but the same cannot be applied directly to mobile apps [38] due to the compatibility issues between the former and the latter's system architectures. Although developed in Java, Android runs in Dalvik virtual machine [13] which is quite different from the Java virtual machine.

Several different testing techniques have been proposed in the literature for testing of mobile apps [2, 3, 12, 37, 39, 41, 50, 55]. However, the majority of these works has focused on testing only one version of a mobile app. Also, several studies [30, 32, 34, 35, 42] have established that manual testing is still prevalent and dominant over automated testing solutions due to the time restrictions imposed on testing, limitations of available research tools, and sometimes, lack of knowledge regarding state-of-the-art approaches. In such a situation, testing mobile applications, which are quite prevalent today, clearly becomes expensive. In particular, due to frequent app releases, testers need to perform more regression testing. This not only increases the testing time but also raises criticality of regression testing. In contrast to other languages such as Java, Android uses an event-based programming model. Several studies have been done to analyze or test the event-driven applications in various domains [23, 28, 44, 56]. However, these event-driven systems are hard to be thoroughly tested since they have to deal with external asynchronous events that exponentially increase their execution paths. Also, this programming style can cause fatal nondeterminism errors which are hard to trade.

Events in Android not only initiate the interaction on mobile screens but also guide the app flow. When we applied the conventional regression test selection approaches to this problem, we faced three major issues. First, Android events are registered not only in Java source but also in layout XMLs. Existing approaches [15] consider only the former, missing the event handling in the layout files. Also, existing approaches ignore the changes in SDK configurations and the permissions in the Android manifest file, which can critically affect the app's functioning. Such a miss makes the underlying tool *not* run relevant testcases on the modified version, potentially missing bugs. Especially in the context of security vulnerabilities

ISSTA 2019, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

prevalent today, missing to analyze meta-data may prove dangerous for the end-user.

Second, in the context of event-aware testing, the output of the state-of-the-art regression test selection (RTS) approaches is not tester-friendly. For instance, RTS leverages change impact analysis techniques which reports certain *methods* affected due to change-impact [8, 43, 45]. But such a piece of information is not useful for testers who perform manual, black box testing (method names and basic blocks would be useful to developers, not testers). While the mapping between methods and test cases can provide information about what test cases to run, due to the nature of these rapidly evolving apps and the changing user requirements, the current set of test cases and the associated map may quickly become irrelevant.

Finally, to add to the misery of the testing team, RTS mentions directly the testcases to be run, which can be too coarse. Instead, in practice, testers would prefer a more granular information in terms of the events (on buttons) to be tested. Note that the testing team is aware only of the testcases and the events (such as `onClick`).

In this paper, we propose the **first activity and event aware regression testing approach** for Android applications. We present a mechanism to study event regressions and assist testers in analyzing them better. Our methodology improves the existing techniques for RTS of Android applications with a richer change-set analyzer that considers Android's event model, configurations, and the various file types. In particular, it leverages a static program analysis approach to determine the effect of regression on the source code. The analysis first finds the entry-points of the app (callbacks/event handlers). It then filters the methods in the call graph corresponding to each event handler/entry point of the app. It also parses the android manifest, app resource, and layout XML files to build the mapping needed; i.e., mapping from activities to user element to events. This mapping assists in finding event-based regressions and also in identifying the app elements to be tested. Further, it addresses the gap in the spectrum of automated mobile testing solutions and manual testing by providing a pictorial representation of the regressions caused due to the change impact and propagation of the code changes.

This paper makes the following contributions:

- We propose a new methodology for activity- and event-aware regression testing of Android apps. The methodology is able to provide a mapping from activities to user elements to events for regression testing.
- We present a richer change-set analyzer that covers code as well as non-code components for regression. It finds a more granular change-set between two versions of an Android app, overcomes the limitations of the existing conventional RTS approaches, and finds event-aware regressions for Android apps.
- We develop a tool named QADroid (Quality Assurance) which implements our new methodology. The tool also assists manual testing by presenting event regressions pictorially.
- We demonstrate the efficacy of our approach by performing an empirical evaluation on 1006 releases of 50 open source Android projects. The results show that QADroid reduces the activity selection by 58% and event selection by 74%, thereby significantly reducing the manual testing efforts.

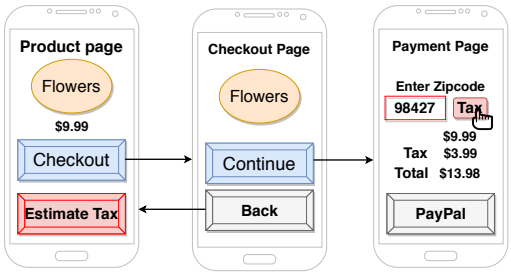

Figure 1: Checkout flow in shopping app example. Tax calculation affects both the checkout page and the product page.

The rest of the paper is organized as follows. Section 2 motivates the need for event-aware regression testing using an app. Section 3 describes our approach QADroid in detail. Section 4 evaluates QADroid and analyzes the results. Section 5 discusses the threats to validity of our empirical study. Section 6 compares and contrasts against relevant related work. We conclude the paper and state directions for future work in Section 7.

2 MOTIVATION

We use a simplified version of a shopping Android app (such as Amazon) to illustrate our approach. The same will be used as a running example throughout. Figure 1 shows the basic flow of the Shopping app checkout and its initiation from the Product page. The events for the buttons on the Product page are registered in XML while events on other pages are registered via Java listeners. The product page has the option of pre-calculating the tax based on the zip code. The same functionality is available on the Payment page. Say the app undergoes a series of changes. A change done on the tax calculation feature on the Payment page will also affect the functionality on the Product page. Let us see how a conventional RTS will fair in dealing with this regression.

Conventional RTS for Android: Figure 2 shows the architecture of a conventional RTS system for Android (such as Redroid [15]). The *Impact Analyzer* module computes the change-set by comparing the two versions of the Android application. The *Coverage Generator* module computes the code coverage for each test case. The information produced by the two modules is fed to the *Test Case Selector* module which results in the set of selected test cases for regression testing. The Impact Analyzer module compares only the Java source files of the two versions and calculates the change-set accordingly.

A conventional RTS will be able to catch this change *provided* all the automatic setup is done, and a coverage map is maintained (which is typically large for large-sized applications and keeps changing with every release). It would be useful, however, to not maintain this costly setup and still be able to highlight the testing to be performed on the Payment page as part of the regression, as well as on the Product page as part of the change-impact. This necessitates lightweight setup to find regressions.

As we can see, conventional RTS provides the set of testcases to run on a code change. However, this is too coarse, as it does not specify what parts of the testcases are really relevant. On the other hand, if we remove the test scripts module from the Conventional

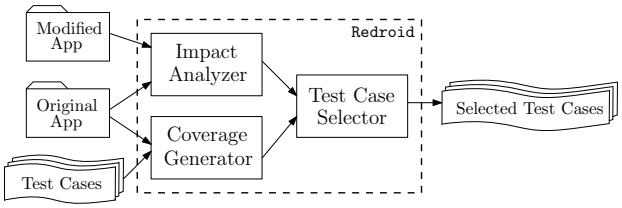

Figure 2: Conventional RTS (e.g., Redroid [15])

RTS architecture, it can provide us with the changed set of methods. Such an information is clearly more fine-grained than a set of testcases; however, this information is too technical for a tester who has never seen the code before. Software industries have separate development and quality assurance teams, and the testers seldom look into the code (rightly so to avoid bias). It would be ideal if the RTS tools can "speak" the language of the testers (in terms of scenarios, events and guilets). This forms our first motivation.

An Android application consists of several components: (i) source class files, (ii) layout-XML files, (iii) configuration files, and (iv) Android manifest. It is worth noting that the existing techniques consider only the Java source files to compute the change-set. In particular, change-sets not reported by the existing RTS approaches are:

(1) Permission(s) removed from the Android manifest.
Consequence: Application fails wherever it requires the missing permission to execute.
(2) Change in configurations; for example, the `minSDKVersion` changed from 10 to 15.
Consequence: The application stops functioning for all the Android phones with SDK versions from 10 to 14, both inclusive.
(3) Change in `onClick()` listener method declared in the layout XML.
Consequence: The application does not behave in the same manner, as the `onClick()` listener is changed.

These are serious misses and ignoring the above change-sets affects the regression testing outcome.

GitHub Study. To understand the extent of usage of XML based events, we conducted a preliminary analysis of 50 open source apps from GitHub [18]. These apps were chosen randomly with the intention of finding at least one XML based listener. We compared the number of XML versus Java-based events; Figure 5 depicts the comparison results. We observe that the apps contain a substantial number of XML based events. This missed opportunity in the existing approaches toward change-sets forms another motivation for our work.

Supporting Examples. Figure 3 shows a change (abstracted from a real-world malware app [10, 58]) in the modified application. The app implements a login activity which takes in the user's credentials and initiates the entry into the Shopping portal. In Android, listeners are defined either directly in the code or in the layout XML file, as shown in the figure. Now, with the change highlighted in Figure 3, when the user clicks the Login button of the activity, the password is sent via SMS (line 5). This change gets **missed** in the existing regression test selection techniques (such as [15]). Figure 4 shows another scenario where a particular permission needed for the proper functioning of the app has been deleted intentionally or accidentally, and the app is ready to ship. Prior work on regression test selection is **unable to identify** this change, which can lead to serious consequences.

We overcome these issues in QADroid, by finding the potential events affected due to regression, and assisting testers to locate them with the help of a pictorial representation of the app. This helps speed up manual testing. We also make the change-set between the two versions more complete, thereby resolving the limitations of the conventional RTS.

3 QADROID: EVENTS-BASED REGRESSION

Android follows an event-based model. All the interactions and actions happening on the mobile screen are initiated by events. Central to our technique is the mapping from events to the underlying Java source code, which enables ready inference of what events to test on regression. QADroid is our regression testing tool for Android applications which employs this technique. Given two versions of an Android application, QADroid's goal is to find the affected set of events. We depict our events-based regression approach in Figure 6. It takes as input the two .apk files corresponding to the two versions of the app, and outputs activity-to-element mapping for the regressed events. To assist testers, it also outputs a pictorial representation of the app's event flow. It consists of five primary modules which we discuss below. Three of them (Event Finder, Call Graph Generator, and Pictorial Flow Generator) can run offline as those need only the first version of the app.

3.1 Module 1: Event Finder

This module takes an apk and generates a mapping of the form Activity → ElementId → Event handler (for instance, ProductPageActivity → CheckoutButton → onClick() listener of CheckoutButton in our running example). We leverage FlowDroid [10] to find application callbacks. The tool considers 181 kinds of events such as `OnMenuItemClick`, `OnDrag`, `OnHover`, `OnSeekBarChange`, `OnScroll`, `OnItemClick`, etc. The overall processing involves three phases. In the first phase, QADroid finds the events registered through Java methods. We build our static analysis of Java classes in the Soot [52] compilation framework. The program analysis performs two tasks. First, it looks for listeners in the Activity classes and the associated element Ids. This provides us the Activity → ElementId → Event handler mapping. Second, it looks for `onCreateView()` method and the associated layout Id. This provides us the Activity → layoutId mapping. In this phase, we can get the mapping of only those events that are registered through Java listeners. In the second phase, QADroid obtains the mapping of the events registered through layout XMLs. QADroid parses the layout XMLs to generate layoutXml → ElementId → Event handler mappings. Some of the callbacks such as `onCreate()`, `onResume()`, `onDestroy()` are framework callbacks and are not related to any particular element in Android. However, in subsequent releases, these callbacks can be overridden, and the methods can also be changed. In case of regression, we do report them but associate

```
1 // LoginLayout.xml  (Original app)
2 <?xml version="1.0" encoding="utf-8"?>
3 <!-- layout elements -->
4 <Button android:id="@+id/loginButton"
5     android:text="Login" android:onClick="loginUser" />
6 <!-- even more layout elements -->
```

```
1 // LoginLayout.xml  (Modified app)
2 <?xml version="1.0" encoding="utf-8"?>
3 <!-- layout elements -->
4 <Button android:id="@+id/loginButton"
5     android:text="Login" android:onClick="sendSMS" />
6 <!-- even more layout elements -->
```

Figure 3: LoginLayout.xml in the original application and the modified version. `onClick` callback is changed from `loginUser()` to `sendSMS()`. Thus, when the user clicks on the Login button of the activity, along with the login id, the password is also sent via SMS.

```
1 // AndroidManifest.xml (Original app)
2 <manifest . . . >
3
4  <uses-permission android:name=
5     "android.permission.CAMERA"/>
6  <uses-permission android:name=
7          "android.permission.WRITE_EXTERNAL_STORAGE"/>
8   <application . . .>
9     . . .
10   </application>
11 </manifest>
```

```
1 // AndroidManifest.xml (Modified app)
2 <manifest . . . >
3
4  <uses-permission android:name=
5     "android.permission.CAMERA"/>
6
7
8   <application . . .>
9     . . .
10   </application>
11 </manifest>
```

Figure 4: AndroidManifest.xml in the original application and the modified version. The change involves removing permission to write to the external storage.

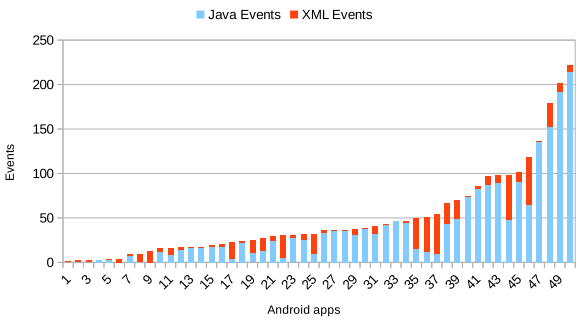

Figure 5: Frequency of events in 50 apps

them with the layoutId of the Activity. The IDs obtained so far such as layoutIds, or the elementIds are references that are converted into numeric identifiers at compile time. Thus we need to resolve these IDs to obtain the layout file name to complete the mapping obtained from the first two phases. These ID references are pointers into the global resource configuration database (`resources.arsc`) included with every Android app. The resource configuration contains a mapping from IDs to resource objects for potentially multiple configurations. In the third phase, we parse `resources.arsc` to generate the layoutId → layoutXml mapping and the ElementId → ElementName mapping. Leveraging the mapping obtained in the third phase, we are finally able to complete the mapping Activity → Element → Event handler. In the static analysis we use ElementIds, whereas while reporting results to the user we use Element names to aid the testers.

Algorithm 1 presents the procedure for generating the mapping Activity → ElementId → Event handler. Lines 1–13 iterates over all the methods in the activity classes of the app and traverses the CFG of the method. Line 4–7 checks if the method is a caller to a

Algorithm 1 Activities → Elements → EventHandler mapping generator

Require: APK of an app, set S_c of callbacks obtained from FlowDroid
Ensure: <Activities → Elements → EventHandler mapping>
1: **for** each Activity class A ∈ app **do**
2: **for** each method m ∈ A **do**
3: Traverse the CFG of m
4: **if** m has call to callback $c' \in S_c$ **then**
5: Traverse CFG of c' in reverse to get EId → EvtHndlr
6: $(A, EId, EvtHndlr)_{java}$ = $(A, EId, EvtHndlr)_{java} \cup (A, EId, c')$
7: **end if**
8: **if** m has call to method `createView` **then**
9: Traverse CFG of *m* in reverse to get layId′
10: (A, layId) = (A, lId) ∪ (A, layId')
11: **end if**
12: **end for**
13: **end for**
14: **for** each layout XML lxml′ ∈ app **do**
15: **for** each element e ∈ lxml′ **do**
16: **if** e has `onClickListener` c in XML **then**
17: (lxml, EId, EvtHndlr) = (lxml, EId, EvtHndlr)∪(lxml′, e, c)
18: **end if**
19: **end for**
20: **end for**
21: (layId, lxml), (EId, EName) = parse(resources.arsc)
22: (A, E, EvtHndlr) = join($(A, EId, EvtHndlr)_{java}$, (A, Id), (layId, lxml), (EId, EName))

callback method. If yes, then it traverses the CFG in reverse direction to obtain Activity → ElementId → Event handler mapping. For the example shown in Figure 1, it generates the mapping of the buttons on the Checkout and Payment pages to their corresponding eventHandlers as the events are registered in Java. Line 8 – 11 checks if the method is a caller to `onCreateView()` method. Generally, `onCreate()` would be the caller to `onCreateView()` method. This helps us in generating a mapping of the form Activity → layoutId. It generates the mapping of Checkout and Payment activities to their respective layout xmls ids. Line 14 – 20 iterate through all the layout XMLs present in the apk and generates

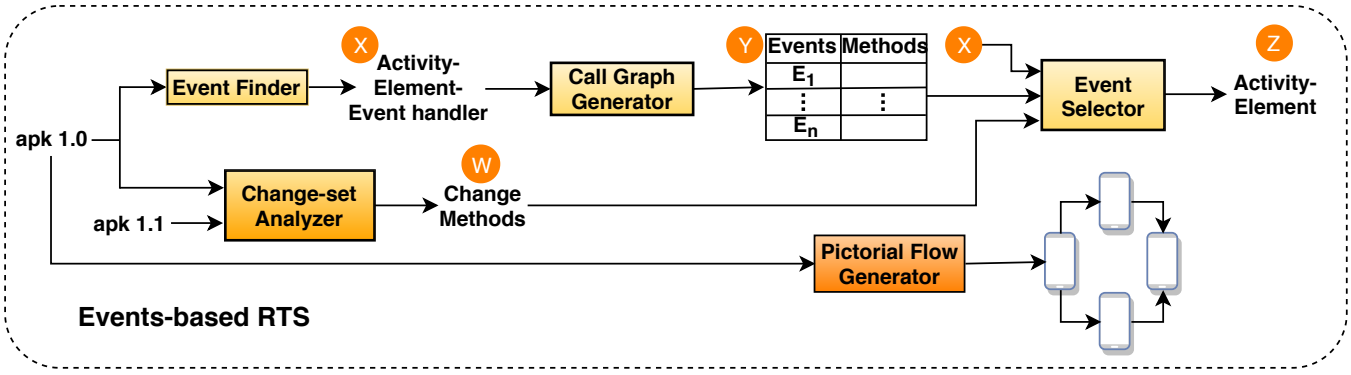

Figure 6: QADroid Architecture

the layoutXml → ElementId → Event handler mapping. It generates the mapping as <ProductPageLayoutXml, CheckoutButton, onClickJavaListener>, <CheckoutPageActivityLayoutXml, ContinueButton, onClickJavaListener>. Line 21 parses the `resources.arsc` file to obtain the mappings layoutXml → layoutId and ElementId → ElementName. The former assist in linking the mapping obtained from Line 4–7 and Line 14 – 20 and generate the complete mapping Activity → Element → Event handler. While the latter assist us in using the ElementId in the traversal analysis and helping us to report the results with Element names. Line – 22 shows the previous step of combining the mapping and generating the complete mapping of Activity → ElementId → Event handler.

3.2 Module 2: Call Graph Generator

Since callbacks are methods, QADroid needs to generate a callgraph. To generate a callgraph for an Android app, we need to first generate an entry point. Unlike Java programs, Android apps do not have a `main()` method. Instead, they overwrite pre-defined methods in Android framework classes. These methods are then called by the OS at runtime according to a specific life-cycle. When generating a precise callgraph, we need to model this life-cycle. Under the hood, FlowDroid generates a dummy `main()` method that emulates how and when the Android OS calls the lifecycle methods. We build upon FlowDroid, obtain the `main()` method and create a callgraph from it. We utilize the Spark tool [33] for callgraph construction. QADroid then performs a BFS traversal, on the call graph taking all the callbacks as the roots of the call graph, one by one. This forms a mapping between callbacks/event handlers and the corresponding methods called (Events → Method mapping). For example, in Figure 1, the Estimate Tax button is linked to the `onClick()` event listener, which in turn, is linked to the `estimatedTax()` method from the call graph.

3.3 Module 3: Change-set Analyzer

This module takes as input the two versions of the Android app and finds the change-set between the two versions. The computation of change-set considers the Java source files, the Android manifest file, and the layout files of the two versions. It constructs the control flow graph (CFG) of all the Java methods and performs a depth-first graph walk on the individual methods from both the versions [49]. For example, if there is a method `findDeviceId()` in both the original and the new versions of the app, it constructs CFG G and G' for the `findDeviceId()` method from both the versions of the app. Given two edges e and e' in the CFGs, if the codes reached by e and e' differ, it marks that method for the change-set between the two versions. QADroid parses the AndroidManifest files of both the versions to extract all the components, permissions, use-features and various other elements along with their attribute information and stores all of it in a map. It then compares the map of the two versions to find the *diff* between the manifest files. The *diff* provides us the advantage of covering the changes in the permission sets, various components, and the configurations needed for the runtime of the application. The changes in the AndroidManifest files are only reported. Similarly, QADroid parses the layout files of both the versions to extract various layout elements and their associated registered events, which are stored in another map for comparison. The changes in the layout XML are leveraged to find the change-set between the events registered through layout XMLs. This covers the changes in the functionality invoked from the layout XML.

Our Change-set Analyzer is functionally (almost) complete as it considers almost all the file-types that are part of the Android APK except the Android native library and the META INF folder which contains the application-specific certificates.

3.4 Module 4: Event Selector

The methods obtained from the change-set analyzer by comparing the two versions of the app can be used to filter the events that would be affected. The events thus obtained help us to traverse back to the particular Activity and element by leveraging the Activity → Element → Event handler map obtained from the Events Finder module. This helps us to deliver outputs in the form of Activity (the name mentioned in the AndroidManifest.xml) and the element name present on the screen. For example, in Figure 1, the Estimate Tax button on the Product page and Tax button on the Payment page are chosen as the events to be tested. However, the output is in the form `ProductPageActivity.xml: Estimate Tax Button` for the former and `PaymentPageActivity.xml: Tax button` for the latter.

3.5 Module 5: Pictorial Flow Generator

In order to help testers localize activities and navigate to activity (mobile screen) in multi-screen apps, QADroid provides an intuitive

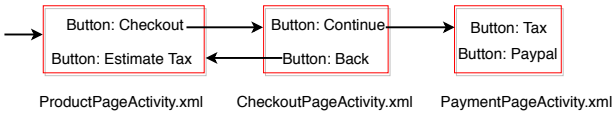

Figure 7: Pictorial representation of Shopping app.

user-interface. The interface makes the results of the event selector module meaningful for manual testing. The GUI presents a pictorial representation of the app's functioning and the flow of various activities. It shows Activities named as per the nomenclature in the AndroidManifest.xml, event elements as defined in the layout XMLs, and the inter-activity flow as determined by the Intent pattern in the call graph methods of the elements' event handlers.

Intents are objects of the android.content.Intent type. They are used in inter-component communication. QADroid looks for one of the following two patterns followed by the `startActivity()` method:

(1) Intent i = new Intent(this, CheckoutPageActivity.class);

(2) Intent i = new Intent(ProductPageActivity.this, CheckoutPageActivity.class);

Our procedure to generate the graphical representation is detailed in Algorithm 2. The pictorial representation of the app needs to have information about the number of components in the app and all the interactivity flows. Line 1 parses the AndroidManifest.xml to get the list of all the components. Lines 2–10 iterates over all the activities in the app and add the activities to the Node list of the app as per Line 3. Lines 4–6 find the first activity of the app. Lines 7–9 add the elements to the respective activities. Lines 11–17 enumerate through Events → Methods mapping and search for the intent pattern. When the intent pattern is found, it creates an edge into the Edgelist. Once Nodes, startNode, and the edgelists are finalized, QADroid plots the app representation using Javascript as listed at Line 18.

Example. The algorithm takes as input the APK of the application, the Events → Method mapping and Activity → elementId → eventHandler mapping obtained from Algorithm 1. In our running example of the Shopping app, it finds three activities as the three components. Product page being the top activity, it marks it as the start activity of the app. So we obtain three components as three nodes. Now on every node, we have various XML elements which we can have as events generating elements. This set can again be leveraged from Algorithm 1. We map these on to the respective nodes. Now, we iterate through all the events and get their corresponding event handler methods. We traverse through these methods in the call graph of the event handler methods and search for the Intent Pattern. If we find any such pattern, we add that edge from Activity 1 to Activity 2 citing the element and the associated event handler. In our example, we obtain an edge from the Product page to the Checkout page due to Checkout button. Similarly, the flow of the app is obtained, and we get a diagram similar to the one shown in Figure 7. Now, if there is a regression on the Payment page due to the changes in `calculateTax` method, the regression results would be easily mapped in the pictorial representation as an aid to the manual testers.

Algorithm 2 Generating app flow

Require: APK, Events → Methods from Algorithm 1, (A,EId,EvtHndlr)from Algorithm 1
Ensure: Pictorial representation of the app

```
1: Parse AndroidManifest.xml to get all the Components C
2: for each activity a ∈ app do
3:     Nodes ← Nodes ∪ {a}
4:     if a has Intent-filter with category LAUNCHER then
5:         topActivity←a
6:     end if
7:     for each (a,eId,evtHndl) ∈ (A,EId,EvtHndlr) do
8:         NodeAttributes ← NodeAttributes ∪ {eId,evtHndl}
9:     end for
10: end for
11: for each a → b ∈ < Events → Methods > do
12:     for all the methods in the event a do
13:         if m contains the Intent pattern then
14:             EdgeSet E ← E ∪ (Activity1, Activity2)
15:         end if
16:     end for
17: end for
18: Plot the graph using Javascript
```

4 EXPERIMENTAL RESULTS

We have evaluated QADroid on an extensive collection of 1006 releases of 50 open source apps from GitHub. Through the experimental evaluation, we aim to establish (A) the significance of the new RTS approach in reducing the manual testing efforts by reporting the selected activities & selected events that alone need to be tested, (B) the time taken, (C) the correctness of our change-set, (D) the usefulness of the output of our Pictorial-flow Generator. We describe the experimental setup and the benchmarks used below.

Experimental Setup. We ran Pūrnadroid using the Android Development Environment, together with IntelliJ IDEA, Eclipse and Android Studio. The study was performed on an Intel Core i7-6700 CPU at 3.40GHz with 32GB of memory with Linux operating system. We use Soot [52] for change-set analysis, FlowDroid [10] for callbacks, and Spark for call graph construction.

Benchmarks. To test QADroid, we analyzed 10K+ GitHub repositories and identified 50 Android projects that consist of multiple releases along with their apks. All these selected projects used GitHub versioning control, which is useful for our change-set validation. Some statistics about the used apps are shown in Table 2. The statistics shown in the table are averaged over the multiple releases of each app. Columns 3-5 show the number of classes, the number of methods and lines of code in each app (averaged over the number of releases), respectively. The app with the most extensive code metrics concerning all the three parameters is Cgeo, the number of classes being 592, the number of methods 5983 and LOC 74680 (data averaged over 67 releases). The smallest app concerning code metrics is AndroidImageViewer, the number of classes being 4, the number of methods 20 and LOC 388 (data averaged over five releases). Columns 6-7 shows the number of activities and the number of events/callbacks in each app (averaged over the number of releases) respectively. We have single screen apps like APK Updater, FastBarcodeScanner and the maximum number of activities are 43 for iNaturalist. The minimum number of events is 4 for NXLoader and Rashr, and the maximum number of events is 242 for Cgeo. On an average, the number of classes across all

Table 2: Some static characteristics and analysis time for the Apps used.

SNo.	App Name	#Class	#Method	LOC	#Activity	#Event	#Commit	#File	#Java	#Manifest	#Layout	#Release	#Time(s)
1.	Aard2	25	309	4114	2.8 3	19.620	318	436	165	38	37	39	12.70
2.	AndroidImageViewer	4	20	388	2	8	22	53	8	1	3	5	6.83
3.	AndroidIssueReporter	12	71	953	3	9	67	59	14	1	4	2	7.31
4.	APK Updater	53	298	4211	1	10	469	656	332	10	45	39	11.95
5.	AraśttaâĎć	34	272	5887	15	72	16	106	34	2	22	12	6.82
6.	Balanduino	14	116	2692	2	18	282	220	73	6	24	9	2.64
7.	BitcoinWallet	98	1034	18346	11	89	1502	4716	1533	137	267	138	55.27
8.	Buendia	428	3135	58486	8	91	2	0	0	0	0	2	39.68
9.	BuildmLearn	119	827	14253	9	31	400	935	354	3	158	4	22.51
10.	Cgeo	592	5983	74680	34	242	5618	9514	4491	23	373	67	42.44
11.	CNode-Material-Design	84	588	5598	17	36	626	675	351	4	108	6	24.28
12.	Diycode	230	1613	20644	10	36	80	163	81	1	22	4	13.17
13.	Emoncms	34	359	5125	6	25	175	384	108	3	28	16	11.40
14.	Evercam	131	1317	15610	24	190	27	137	39	5	31	6	40.99
15.	FastBarcodeScanner	13	89	1648	1	10	2	18	0	0	0	2	9.21
16.	FetLife	209	1978	22390	22	85	213	511	180	1	58	2	41.06
17.	Forecastie	8	62	909	2	9	142	198	27	5	5	9	5.65
18.	Hentoid	66	514	7604	12	60	960	1637	755	16	154	26	13.44
19.	iNaturalistAndroid	152	2128	40459	43	166	806	2240	888	144	244	146	33.56
20.	IOTA Wallet	106	725	9503	4	19	17	33	16	0	0	3	27.57
21.	JChat	138	1349	21714	32	126	466	2619	1052	15	496	16	19.43
22.	Kegbot	147	1343	19315	17	110	201	1094	536	14	117	15	44.85
23.	LocationReportEnabler	5	36	519	1	8	34	61	13	3	6	6	6.94
24.	MaterialIntro	17	286	3244	4	12	229	223	69	3	28	6	7.08
25.	Mattermost	22	136	1933	4	16	173	101	34	5	7	14	14.87
26.	MetaWear	32	253	5639	4	18	37	1014	365	14	151	21	5.80
27.	MLManager	14	98	1525	4	42	271	363	87	7	30	13	22.36
28.	MTG Familiar	79	845	24159	2	27	1018	4574	688	14	200	24	20.38
29.	NXLoader	8	30	440	2	4	20	49	13	0	5	3	7.58
30.	ObscuraCam	47	496	7805	6	60	42	136	78	3	11	5	13.63
31.	OneBusAway	243	2667	41301	28	112	1659	2680	1144	20	145	42	24.74
32.	OpenTripPlanner	43	538	12025	4	20	10	16	11	0	0	3	14.44
33.	OTP Authenticator	13	85	1484	3	19	24	31	5	0	1	4	9.45
34.	Pandora	94	660	8954	3	14	30	138	87	3	12	8	9.83
35.	Paperwork	24	162	3370	4	32	27	155	31	1	7	5	5.51
36.	Popular Movies	39	318	3063	3	13	29	170	66	1	22	3	3.38
37.	PrimitiveFTPd	67	687	10974	7	28	257	369	203	12	11	20	20.61
38.	Rashr	61	636	11739	3	4	269	830	199	6	132	10	13.00
39.	Reddit	71	377	3840	4	7	208	369	255	4	27	11	13.94
40.	Resplash	64	525	9398	14	79	77	333	144	9	35	18	61.71
41.	RxJavaApp	34	397	6172	4	10	67	75	22	1	9	5	20.39
42.	SkyTube	84	546	8845	4	15	1176	1240	426	7	58	14	21.13
43.	Slide	152	1026	25660	42	186	1615	3247	841	7	486	11	39.34
44.	SteamGifts	106	857	10002	10	31	562	2039	1283	28	249	95	17.53
45.	StreetComplete	299	2361	25903	3	41	1321	5800	1549	7	204	46	16.59
46.	TheBlueAlliance	375	2549	32560	19	91	2493	5449	3580	25	373	64	20.90
47.	TvAppRepo	37	240	3553	6	5	38	135	56	2	8	11	30.64
48.	TwrpBuilder	47	219	3793	9	39	54	228	90	3	39	35	21.76
49.	Wallabag	58	638	7661	7	44	853	759	412	14	57	26	13.49
50.	WhatAndroid	122	987	14049	36	67	631	2153	438	13	271	15	11.82
	Average	**99**	**856**	**12883**	**10**	**50**	**513**	**1183**	**465**	**13**	**96**	**22**	**19.632**

apps (averaged over all releases) is 99. Similarly, the number of methods is 856, LOC is 12883, the number of activities is 10, and the number of events is 50 per app (averaged over all apps all releases). Columns 8-9 show the total commits and the total number of file changes that each app undergoes across all the releases. Columns 10-12 show the total number of changes in the Java, Manifest and the layout files across all the releases. Column 13 shows the total number of releases considered per app; on an average, we consider 22 releases per app. Finally, Column 14 shows the time taken by QADroid on an average to analyze each app.

4.1 Events-based regression

We first discuss the selection of Activities separately, followed by the selection of Events.

Selection of Activities. Figure 8 shows the percentage of activities selected for regression testing averaged over multiple releases. The apps are arranged in the alphabetical order as maintained in Table 2. The activities-based regression selection reports 41.96% activity selection for regression testing averaged over 1K+ releases of 50 apps. In other words, for an app having 10 activities (mobile screens) and changes spanning across 22 releases, QADroid would help the tester to narrow down the regression testing from 10 activities to 4 particular activities.

QADroid reports 100% activity selection for apps NXLoader and LocationReportEnabler. It is noteworthy that LocationReportEnabler is a single screen app. So, any change in the relevant files results in the selection of that single activity for regression testing. NXLoader on the other hand has two activities and it undergoes changes in both the Activity classes. However, for another single screen app FastBarcodeScanner, QADroid reports 0% activity selection. On observing the change-set, we found that the app had two releases with changes across 18 files, but there was no change in Java, Manifest and layout XML (see Table 2). Similarly, QADroid reports 0% activity selection for Buendia which had four activities. This is a unique app where we found no file changes across the releases: there was a separate release with two commits where only new tag names were introduced. We kept this app to cover the corner case of no change in the app at all. Among the multiple screen apps, the highest activity selection is reported for CNode-Material-Design 96.34%. The app has 17 activities (mobile screens). The app had changes across six releases with 351 Java changes, 108 layout XML changes averaging to 60 Java files changes, and 18 layout file changes per release. On careful observation of the change-set, we found continuous development in all the modules. On the other hand, the app with the maximum reported changes is Cgeo with 4491 Java changes and 373 Manifest changes across its 67 releases. We studied this closely and observed that the app has many activities for third-party integrations and does not involve changes in those activities beyond a particular release; for example, TwitterAuthorizationActivity for Twitter authorization, various activities for map integration like GoogleMapActivity, ForgeMapActivity, StaticMapActivity. We seldom see changes in these activities. Hence these activities are not chosen for activity selection amounting to 62% activity selection.

Overall, the above results indicate that manual testers can avoid testing over 58% of the activities (mobile screens), thereby saving a considerable amount of efforts and time. However, there can be multiple events on each activity/screen. So, we next narrow down and pinpoint the element to be tested and the type of events (such as onClick(), onLongClick(), onKeyDown(), onTouch(), and so on) that need to be tested.

Selection in Events. Figure 9 shows the percentage of events selected for regression testing averaged over multiple releases. The events-based regression selection reports 25.54% event selection for regression testing averaged over 1K+ releases of 50 apps. In other words, for an app having 50 events and changes spanning across

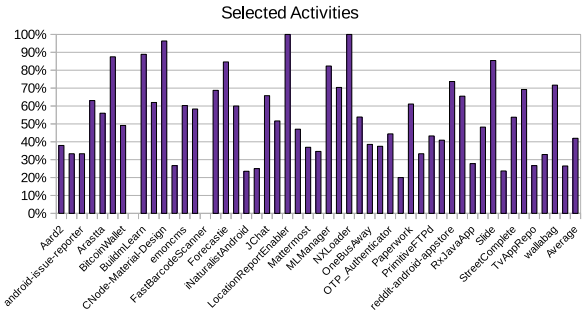

Figure 8: Selected Activities.

22 releases, QADroid would help the tester to narrow down the regression testing from 50 events to 13 particular events.

For the single screen apps like LocationReportEnabler and NXLoader having two screens, where QADroid reported 100% activity selection, events based selection helps to bring down the numbers slightly. We do not see much gains in NXLoader, where the events selection reports 85.71%. The app has four events, and it chooses three or four events based on the release. For LocationReportEnabler, we see a higher gain (43.18% event selection). The app has eight events, and about four events are selected. For the apps like FastBarcodeScanner and Buendia, where the activity selection was 0%, event selection also reports the same which is rightly so. Among the multiple screen apps, the highest activity selection is reported for CNode-Material-Design, event selection helps to bring it down from 96% to 69.14 %. Cgeo reports 45.87% event selection. Empirically, it chooses 111 out of 242 events for regression testing, thereby saving efforts and time over testing of 131 events. Mostly, we see the trend observed in activity selection reflects in event selection as well, with the event selection numbers slightly less than the activity selection. However, we do see an exception here. The app TvAppRepo reports 26.67% activity selection and 40.91% event selection. On studying the app, we see that the events are concentrated in one Activity "AdActivity". For instance, in the first release of the app, the app has 7 activities and 18 events. However, the single activity AdActivity has 9 of these 18 events. In the change-set when the second release is out, we see changes in 3 activity classes including AdActivity, thereby making the activity selection as 42% and the event selection 9/18 (that is, 50%). Other apps such as IOTAWallet also show a non-uniform concentration of events across the activities, however with TvAppRepo we see this feature quite predominantly.

We can also observe that QADroid selects less than 70% activities for 80% of the apps, and less than 50% of events for 86% of the apps. Overall, the above results indicate that manual testers can avoid testing over 74% of the events, thus saving a considerable amount of efforts and time.

4.2 Change-set Analysis

An Android project consists mainly of three folders *main*, *Test* and *androidTest*. *Test* folder is a placeholder for keeping JUnit tests and *androidTest* is a placeholder for keeping the Android instrumented tests. Since *main* consists of the application's business logic and

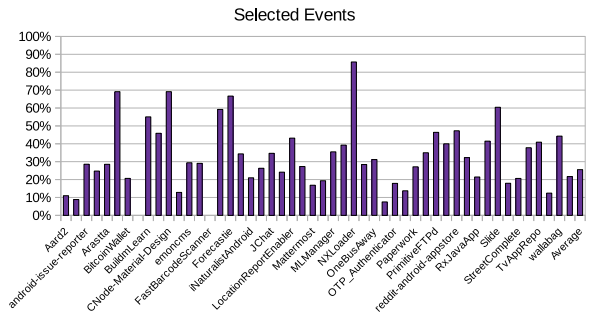

Figure 9: Selected Events.

resources, we leverage the artifacts in this for our analysis. We get the *main* package from Android Manifest.xml to identify the change-set. As reported in Table 2, we consider the changes in the following files - layout files, manifest files, Java files, as these are the only files that contribute to the functional aspect of the application. Besides, QADroid can report changes in other files as well. However, it is on the coarser granularity (file-level). On an average, we see that the apps undergo 465 Java file changes, 13 Manifest file changes and 96 layout file changes (Table 2).

Change-set Validation. We manually validate all the change-sets with the changes reported by the GitHub Compare View utility between the releases. The GitHub Compare View utility compares View URLs and generates URLs of the following form:

`http://github.com/U/R/compare/v(i)...v(i+1)`

- `U` - User's GitHub repository
- `R` - Android's app repository URL
- `v(i)` - release number of older version
- `v(i+1)` - release number of newer version

We found that our change-set evaluation is complete for the data set used.

4.3 Time taken

Column 14 in Table 2 shows the time taken by QADroid on all the apps. The time for each app is averaged across the releases of the app. We find that on an average, QADroid takes just 19.63 seconds to find the events to be regression tested. The maximum average time across the apps is obtained for BitcoinWallet, 55.27 seconds. Interestingly, the maximum time across a release is also reported for BitcoinWallet which is around 212 seconds. As the analysis done by the QADroid is completely static, we believe it is quite reasonable.

4.4 Pictorial Flow Generator

We generated the pictorial flow of the apps for at least one version of every app. We can depict the flow results easily for apps having a fewer number of activities. As the number of activities (mobile screens) become huge, the output of the generator output along with the elements become clumsy. So, we generate a web view which only highlights the activities and its flow. Pictorial Flow Generator shows the event generating elements if the number of event-generating elements is less than two. If the elements are more in number, we provide a hover utility that displays the event generating elements. For example in Figure 10 we show the pictorial representation of android-issue-reporter. The number of event generating elements in the *air_activity_issue_reporter.xml* is four (more than two). So, we do not show all the elements. On hovering on the `Browse Elements` button, the event generating elements are shown.

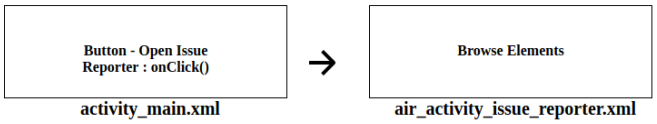

Figure 10: Pictorial Event Flow for Android Issue Reporter.

5 THREATS TO VALIDITY

We discuss the threats to validity of our study according to the Cook and Campbell categorization [14].

Internal Validity. This refers to the causality relationship between treatment and outcome. In our study, we find the events to be regression tested, relying on a prior work FlowDroid for callback calculation. Similar to other static analysis tools, FlowDroid has certain inherent limitations which may affect the internal validity of our study. For instance, FlowDroid resolves reflective calls only if their arguments are string constants, which is not always the case. Also, it maintains an exhaustive list of 181 kinds of callbacks. Thus, unsoundness may arise in case the Android lifecycle contains callbacks that FlowDroid is not aware of, or through native methods modeled incorrectly by its rules.

External Validity. This refers to the generalizability of our findings. To ensure that our subjects are representative of the population of Android apps, we analyzed a large, heterogeneous dataset of Android open-source apps available on GitHub (belonging to different domains and having different sizes). We browsed through 10K+ GitHub repositories and chose a dataset of 1006 releases of 50 open source apps from GitHub. The apps were selected without any bias and based on only one pre-condition that multiple versions of apps are available along with the *apk*s and the source code. Also, few of the apps used in our evaluation are available on Google Play Store [21] with huge user downloads. So, we believe that our dataset still allows us to generalize our results, but that further studies are needed to extend our findings to all Android apps.

Construct Validity. It deals with the relation between theory and observations. The goal of our study was to reduce the manual testing efforts by regression activity and event selection. We were cautious when collecting and presenting the results and drawing conclusions based on them. We show the results and the savings in terms of activity and event selection and give it as an indicative measure of the amount of savings. However, we cannot assert that the numbers in these results directly reflect the exact magnitude of savings in time or manual efforts as different events can take different times in execution. Some events may execute faster while some might take longer time. It also depends upon the hardware and

its load during measurements. The results are simply an indication of the savings but do not reflect the actual time/efforts saved.

6 RELATED WORK

To the best of our knowledge, there is no prior work on techniques to identify the regression event selection for Android apps. Android apps are event-driven, and several studies have been performed on test input generation (event generation) for testing of Android apps. Based on their approach for generating inputs to mobile apps, the existing studies are broadly classified into three categories: random/fuzz testing, which generates random inputs/events to the app; systematic testing, which systematically tests the app by executing it symbolically; and model-based testing, which generate events according to certain model such as GUI model of the app, finite state machines, etc. We discuss these below. We later also contrast our change-set analyzer with the existing works on change impact analysis.

Random testing. To facilitate random UI testing, the Android platform introduced a fuzz testing tool Monkey [7] that generates a sequence of random events to aid in the testing of Android apps. Amalfitano et al. [2] described a crawling-based approach to generate random but unique test inputs. Several approaches build upon Android Monkey to come up with better random testing techniques. Hu and Neamtiu [24] leveraged Android Monkey in a random fashion to generate GUI tests. Dynodroid [36] provides techniques to improve on Android Monkey's performance by incorporating several heuristics. However, these works suffer from the following limitations – redundant events, inefficiency in generating inputs that require human intelligence, incompetence in exploring new behavior of the app, unable to generate system events and are limited to only GUI events.

Model-based testing. Several approaches [3, 11, 12, 22] have focused on the testing of Android apps via GUI models. GUIRipper [3] [6] dynamically analyses the application's GUI with the aim of obtaining sequences of events fireable through the GUI widgets. It maintains a state machine model of the GUI (GUI tree) and follows a depth-first exploration strategy to test the app. Following the same exploration strategy, ORBIT [54] is a grey-box model creation technique that creates a GUI model of the app for testing. Azim et al.Azim13 builds the app model by employing a static taint analysis technique and utilize this model for automated exploration of an app's Activities for testing. The EXSYST tool [22] utilizes runtime feedback to continually refine models and results in increasing the code coverage of the given app. The quality of the GUI models defines the effectiveness of these approaches which in practice are abstract and may not cover the entire behavior of the apps.

Advanced testing. Several works [5, 40] have applied symbolic execution to generate inputs to Android apps. Jensen et al. use symbolic testing to generate event sequences that reach specified target locations [29]. These approaches can generate highly specific inputs and complex event sequences, but due to the heavy instrumentation of both the Android framework and the app, they suffer from scalability issues. TrimDroid [39] generates a subset of event sequences by leveraging constraint solver thereby achieving a considerable coverage. Sapienz [37], on the other hand, minimizes the length of event sequences by exploring test sequences leveraging a search-based testing methodology. Tools AppDoctor [25] and EHBDroid [51] invoke the event handlers to simulate events. The scope of AppDoctor tool is limited as it considers only 20 event types and focuses on finding app crashes due to specific bugs. In contrast, EHBDroid handles 58 kinds of callbacks and is efficient towards automated GUI testing of Android apps.

Compared to existing works on mobile app testing, QADroid is different in the sense that it does not generate events; instead, it highlights the events whose behavior has changed and need testing, thereby aiding in efficient and effective manual testing. Also, most of the existing works focus on testing one version of the app while QADroid, in contrast, helps in identifying regressions introduced as part of incremental changes. Though our implementation is confined to Android apps, our approach can be extended to other event-driven systems.

Change impact analysis. It is an important and well-studied problem in software evolution [8, 17, 43, 45, 57]. Chianti [45] perform pairwise comparisons of the abstract syntax trees of the classes in the two project versions. However, it works only on Java programs. In contrast, QADroid compare the CFG of the methods in the classes of the two versions of the same Android app using Dejavu algorithm [49] which uses a notion of statement-level coverage. Ekstazi [19] is a lightweight and scalable Java library for regression testing. It calculates the change-set at the file level. It uses checksums (calculated as a hash of the file contents) to find changes at the file level. QADroid, on the other hand, finds the changeset at the method level and is thus resistant against simple non-semantic changes that may affect the checksum. Very little work has been done on change impact analysis for Android apps. To the best of our knowledge, Redroid [15] is the first approach towards regression test selection for Android applications which is based on the testcases-to-methods mapping. We find that it covers only the changes in source code. To its contrast, QADroid performs regression event selection using the events-to-methods mapping. QADroid covers changes in the code as well as the non-code components and thus has more complete coverage. QADroid is the first tool that works holistically on the complete Android apk.

7 CONCLUSIONS AND FUTURE WORK

We presented QADroid, which performs activity- and event-aware regression selection for Android applications. Its pictorial event-flow representation helps the manual testers to visualize the testing points in a tester-friendly manner. QADroid finds a more complete functional diff (change-set) between two or more versions of the same Android application. It considers the Java source codes, configurations, and also the event bindings in layout XML to find the change-set. The completeness of the change-set helps us report complete diff and, in turn, perform better event selection. QADroid reduced the activity selection by 58% and event selection by 74% compared to conventional RTS, considerably reducing the efforts of the manual testers. This work can be extended by including a fault localization module to map the failed scenarios on the particular elements on mobile to the underlying change-set information. This would help reduce the software release lifecycle.

REFERENCES

[1] Google Inside AdWords. 2018. Building for the next moment. https://adwords.googleblog.com/2015/05/building-for-next-moment.html accessed: 2018-10-10.

[2] D. Amalfitano, A. R. Fasolino, and P. Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 252–261. https://doi.org/10.1109/ICSTW.2011.77

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 258–261. https://doi.org/10.1145/2351676.2351717

[4] Amazon. 2018. Amazon android app. https://play.google.com/store/apps/details?id=com.amazon.mShop.android.shopping

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666

[6] Android. 2018. Android Developer. https://developer.android.com/about/index.html Accessed: 2018-01-21.

[7] Android. 2019. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey Accessed: 2019-01-10.

[8] T. Apiwattanapong, A. Orso, and M. J. Harrold. 2005. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 432–441. https://doi.org/10.1109/ICSE.2005.1553586

[9] Apkpure. 2018. Apkpure. https://apkpure.com/ accessed: 2018-10-10.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[11] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[12] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 238–249. https://doi.org/10.1145/2970276.2970313

[13] Buzbee B Cheng B. 2010. A JIT compiler for AndroidâĂŹs Dalvik VM. https://dl.google.com/googleio/2010/android-jit-compiler-androids-dalvik-vm.pdf accessed: 2018-10-10.

[14] T.D. Cook and D.T. Campbell. 1979. *Quasi-experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin. https://books.google.co.in/books?id=BFNqAAAAMAAJ

[15] Quan Chau Dong Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. 2016. Redroid: A Regression Test Selection Approach for Android Applications. In *The 28th International Conference on Software Engineering and Knowledge Engineering, SEKE 2016, Redwood City, San Francisco Bay, USA, July 1-3, 2016*. 486–491. https://doi.org/10.18293/SEKE2016-223

[16] Forbes Entrepreneurs. 2018. How to Build a Mobile-First Company. http://www.forbes.com/sites/tomtaulli/2013/03/21/how-to-build-a-mobile-first-company/ accessed: 2018-10-10.

[17] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. 2012. Integrated impact analysis for managing software changes. In *2012 34th International Conference on Software Engineering (ICSE)*. 430–440. https://doi.org/10.1109/ICSE.2012.6227172

[18] GitHub. 2018. GitHub. https://github.com/ accessed: 2018-10-10.

[19] M. Gligoric, L. Eloussi, and D. Marinov. 2015. Ekstazi: Lightweight Test Selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 713–716. https://doi.org/10.1109/ICSE.2015.230

[20] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. 2014. Regression Test Selection for Distributed Software Histories. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag New York, Inc., New York, NY, USA, 293–309. https://doi.org/10.1007/978-3-319-08867-9_19

[21] Google. 2018. Google Play Store. https://play.google.com/store/apps

[22] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-based System Testing: High Coverage, No False Alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 67–77. https://doi.org/10.1145/2338965.2336762

[23] S. Hong, Y. Park, and M. Kim. 2014. Detecting Concurrency Errors in Client-Side Java Script Web Applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 61–70. https://doi.org/10.1109/ICST.2014.17

[24] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 77–83. https://doi.org/10.1145/1982595.1982612

[25] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/2592798.2592813

[26] Business Insider. 2018. Facebook is officially a mobile-first company. https://www.businessinsider.in/Facebook-is-officially-a-mobile-first-company/articleshow/49680725.cms accessed: 2018-10-10.

[27] Smart Insights. 2018. Mobile Marketing Statistics compilation. https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ accessed: 2018-10-10.

[28] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. 2015. Stateless Model Checking of Event-driven Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 57–73. https://doi.org/10.1145/2814270.2814282

[29] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated Testing with Targeted Event Sequence Generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 67–77. https://doi.org/10.1145/2483760.2483777

[30] M. E. Joorabchi, A. Mesbah, and P. Kruchten. 2013. Real Challenges in Mobile App Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 15–24. https://doi.org/10.1109/ESEM.2013.9

[31] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective Regression Test Case Selection: A Systematic Literature Review. *ACM Comput. Surv.* 50, 2, Article 29 (May 2017), 32 pages. https://doi.org/10.1145/3057269

[32] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. 2015. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102609

[33] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer-Verlag, Berlin, Heidelberg, 153–169. http://dl.acm.org/citation.cfm?id=1765931.1765948

[34] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How Developers Detect and Fix Performance Bottlenecks in Android Apps. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 352–361. https://doi.org/10.1109/ICSM.2015.7332486

[35] M. Linares-VÃąsquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. 2017. How do Developers Test Android Applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 613–622. https://doi.org/10.1109/ICSME.2017.47

[36] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[37] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[38] Roberto Minelli and Michele Lanza. 2013. Software Analytics for Mobile Applications–Insights & Lessons Learned. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE Computer Society, Washington, DC, USA, 144–153. https://doi.org/10.1109/CSMR.2013.24

[39] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 559–570. https://doi.org/10.1145/2884781.2884853

[40] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing Android Apps Through Symbolic Execution. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5. https://doi.org/10.1145/2382756.2382798

[41] K. Moran, M. Linares-VÃąsquez, C. Bernal-CÃąrdenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 33–44. https://doi.org/10.1109/ICST.2016.34

[42] K. Moran, M. L. VÃąsquez, and D. Poshyvanyk. 2017. Automated GUI Testing of Android Apps: From Research to Practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 505–506. https://doi.org/10.1109/ICSE-C.2017.166

[43] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging Field Data for Impact Analysis and Regression Testing. In *Proceedings of the*

9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11). ACM, New York, NY, USA, 128–137. https://doi.org/10.1145/940071.940089

[44] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/2509136.2509538

[45] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 432–448. https://doi.org/10.1145/1028976.1029012

[46] International Data Corporation (IDC) Research. 2018. Android dominating mobile market. http://www.idc.com/promo/smartphone-market-share/ accessed: 2018-12-10.

[47] Brian Robinson, Mithun Acharya, and Xiao Qu. 2012. Configuration Selection Using Code Change Impact Analysis for Regression Testing. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM) (ICSM '12)*. IEEE Computer Society, Washington, DC, USA, 129–138. https://doi.org/10.1109/ICSM.2012.6405263

[48] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *IEEE Trans. Softw. Eng.* 22, 8 (Aug. 1996), 529–551. https://doi.org/10.1109/32.536955

[49] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol.* 6, 2 (April 1997), 173–210. https://doi.org/10.1145/248233.248262

[50] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 864–880. https://doi.org/10.1145/2983990.2984011

[51] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI Testing for Android Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 27–37. http://dl.acm.org/citation.cfm?id=3155562.3155570

[52] soot. 2018. Soot. https://github.com/Sable/soot accessed: 2018-01-21.

[53] Uber. 2018. Uber app. https://www.uber.com

[54] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 250–265. https://doi.org/10.1007/978-3-642-37057-1_19

[55] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia (MoMM '13)*. ACM, New York, NY, USA, Article 68, 7 pages. https://doi.org/10.1145/2536853.2536881

[56] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. 2013. SimRacer: An Automated Framework to Support Testing for Process-level Races. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 167–177. https://doi.org/10.1145/2483760.2483771

[57] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2012. FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 40, 4 pages. https://doi.org/10.1145/2393596.2393642

[58] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 95–109. https://doi.org/10.1109/SP.2012.16