CAT: Change-focused Android GUI Testing

Chao Peng
University of Edinburgh
Edinburgh, United Kingdom
chao.peng@ed.ac.uk

Ajitha Rajan
University of Edinburgh
Edinburgh, United Kingdom
arajan@ed.ac.uk

Tianqin Cai
Bytedance Network Technology
Beijing, China
caitianqin@bytedance.com

***Abstract*—Android Apps are frequently updated, every couple of weeks, to keep up with changing user, hardware and business demands. Correctness of App updates is checked through extensive testing. Recent research has proposed tools for automated GUI event generation in Android Apps. These techniques, however, are not efficient at checking App updates as the generated GUI events do not prioritise updates, and instead explore other App behaviours.**

We address this need in this paper with CAT (Change-focused Android GUI Testing). For App updates, at the source code or GUI level, CAT performs change impact analysis to identify GUI elements affected by the update. CAT then generates GUI event sequences to interact with the affected GUI elements.

Our empirical evaluations using 21 publicly available open source and 2 commercial Android Apps demonstrate that CAT is able to automatically identify GUI elements affected by App updates, generate and execute GUI event sequences focusing on change-affected GUI elements. Comparison with two popular GUI event generation tools, DroidBot and DroidMate, revealed that CAT was more effective at interacting with the change-affected GUI elements. Finally, CAT was able to detect previously unknown change-related bugs in two open source Apps. Developers of the commercial Apps found CAT was more effective than their in-house GUI testing tool in interacting with changed elements and faster at detecting seeded bugs.

***Index Terms*—software testing, android, graphical user interface, program analysis**

I. Introduction

Close to 3 million Apps are available on the Google Play store for Android users. These Apps are frequently updated, typically every week or two, to keep up with changing user, hardware and business demands. To ensure security and correctness, updates in Apps need to be tested thoroughly to ensure changes and existing functionality work as expected.

Several different testing techniques have been proposed in the literature for testing mobile Apps [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. Majority of existing work focuses on testing only one version of a mobile App. For updates in Apps, existing test generation work is not effective since it is not focused on updates and may not even exercise them. There is existing body of work on regression test selection [11], [12], [13], [8], [14], [15] - from an existing suite of tests, regression test selection chooses a subset of tests that exercises updates in an App. QADdroid [11] is the only tool in literature that considers changes and their impact at the GUI level when selecting regression tests. Regression test selection techniques only select tests from an existing test suite, they do not generate new tests that exercise changes. None of the existing techniques support GUI test generation targeting updates in Android Apps.

In this paper, we propose a novel approach for generating GUI events targeting App updates. We support updates to source code and the graphical user interface (GUI). We design and implement a framework named CAT (Change-focused Android GUI Testing) that first gathers GUI elements impacted by the update (referred to as target GUI elements) and then generates GUI events (or test inputs) to exercise these GUI elements. For updates in the source code, CAT's first step of gathering affected GUI elements entails analysing and tracing impact of changes in the source code to target GUI elements. To do this, CAT builds a map relating source code functions to GUI elements and associated Android `Activities` from the App package file (APK). For updates at the GUI level, CAT gathers all the target GUI elements affected by the update and identifies the `Activities` associated with them. For the next step of GUI event generation, CAT builds on an existing model-based android testing tool, DroidBot [16], to generate events interacting with the target elements. Additionally, CAT generates event sequences, rather than single events, to interact with the target element. Using event sequences allows for more rigorous testing, exercising the target GUI element in different contexts (sequence of events leading to it).

We evaluate usefulness and effectiveness of CAT in testing App updates with a dataset of 21 open source from the F-Droid App market and 2 popular commercial Android Apps (`TikTok` and `Huoshan`) developed by ByteDance. We compare performance of CAT against two state of the art model-based GUI testing tools for Android, DroidBot (DB) and DroidMate (DM) [16], [2]. In addition, developers at Bytedance compared CAT to their in-house GUI testing tool. We generate 1000 input GUI events for the open source Apps and 2000 inputs for the commercial Apps with all three tools.

CAT was able to trace updates to target GUI elements and generate event sequences interacting with them in all open source and commercial Apps. We found CAT interacted with the target GUI elements more frequently than DB and DM – 74 interactions per App on average for CAT versus 5 for DB and 3 for DM. We found events generated by CAT interact with the target GUI elements sooner and more reliably than other tools owing to CAT's prioritisation of target elements in event generation. Finally, CAT was able to reveal previously undetected bugs in two Apps in the dataset – `World Weather` and `BeeCount`. DM did not reveal bugs in any of the Apps, while DB revealed a bug in the `World Weather` App but not `BeeCount`. We find the combination of target element priority in event generation along with rigorous target element interaction with event sequences makes CAT an effective test

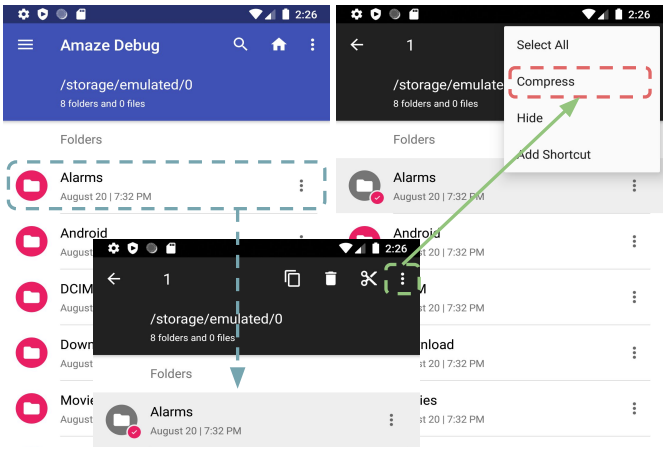

Fig. 1: Sequence of input events to interact with `Compress` option in Amaze File Manager App

generation tool for App updates.

Unsurprisingly, no change-related bugs were found in the released versions of the commercial Apps as these are extensively tested before release. To compare CAT against the company's in-house testing tool, App developers seeded a fault in the change impacted source code for TikTok. They found CAT was 12 times faster than their in-house tool in detecting the crash triggering bug. In summary, the main contributions in this paper are,

1) Novel GUI input generation technique targeting Android App updates, with support for change impact analysis from source code to GUI level.
2) Empirical evaluation comparing performance of CAT against DB and DM using 21 open source Android Apps and 2 commercial Apps, `TikTok` and `Huoshan`.

II. Motivation

In this section, we highlight the need for change-focused test input generation with a motivating example – an actively-maintained Android App, Amaze File Manager[1], with more than 3,000 stars on GitHub. Amaze File Manager is used for Android filesystem management. In addition to basic file operations such as copy and paste, it also supports compression, encryption and cloud service synchronisation. We use the latest version, 3.4.3, of Amaze File Manager in this paper. Commit history for the project revealed that the *ZipService.java* source file was updated in the latest version. This file contains the implementation for the file compression functionality.

Changes in *ZipService.java* affect dependent files, *ProcessViewerFragment.java* and *MainActivityHelper.java*. These two Java files are linked to the `compress` option in the main menu, according to the layout file *menu/contextual.xml* used by the *MainActivity* toolbar. Thus, the target GUI element affected by the update is the `compress` option. We checked this by clicking the `compress` option, and found it exercised the changed code in *ZipService.java*.

Figure 1 illustrates the sequence of input events to interact with the target GUI element–`compress` option: long clicking one item in the file list, clicking the menu icon, and then clicking the `compress` option. The `compress` option is included in the toolbar layout of the main `Activity` and this layout only appears after long clicking a file or folder.

When we run existing test input generation tools such as DM [2] and DB [16] on this App, they explore the GUI model without prioritising interactions with the *Compress* option. As a result, these tools may not even exercise the target GUI elements. We monitored events generated by DB as an example, and found it entered the final view in Figure 1 after 59 GUI events. It takes a further 442 GUI events to click the `compress` option (as it explores the `Add ShortCut` option in the menu first) which finally triggers the updated implementation in *ZipService.java*.

The uncertainity observed with existing testing tools in exercising target GUI elements raises the need for a GUI test generation tool that prioritises interactions with these GUI elements. We address this need with CAT.

III. Background

Before we discuss our approach in detail, we briefly introduce basic concepts in Android App development and testing.

A. Android Apps

Android Apps are commonly programmed using Java or Kotlin that are compiled to Java bytecode. Native code can also be included to boost performance. Java bytecode is translated to Dalvik bytecode and stored in a machine executable file in .dex format. Android SDK tools bundle Dalvik bytecode, native code (whenever present) along with any data and resource files into an APK, an Android package, which is an archive file with a `.apk` extension. The APK file is all that is needed to install the App on Android devices.

To build the APK file, an Android project uses the following components: (i) source class files containing source code implementing classes and functions for the App, (ii) layout-XML files which defines the GUI layout of all the `Activities`, and (iii) Android manifest which appears in the App root folder as *AndroidManifest.xml* and describes essential information about the App – package name of the App (used to locate the source code), lists of components in the file, user permissions required, hardware and software features used, API libraries needed.

In the rest of this section, we describe terms and concepts in Android App development used in the rest of this paper.

B. Terminology

An `Activity` implements a window or screen in the App containing various `GUI elements`, such as buttons and text areas. Developers can control the behaviour of each `Activity` by implementing appropriate callbacks for each life-cycle phase (i.e., created, paused, resumed, and destroyed). `Activities` are first declared in the *AndroidManifest.xml* file and implemented as Java classes in the source code folder.

`GUI elements` (also referred to as `Views` or `Widgets`) are the basic building blocks for user interactions, such as textboxes, buttons and containers of other `GUI elements`. `Views` can be associated to `Activities` either in the source code or defined in the XML layout files. An `Activity` uses a

[1]https://github.com/TeamAmaze/AmazeFileManager

GUI registration function, setContentView(), whose parameter is the identity of a layout file to include `Views` defined in that layout. For instance, the menu as shown in Figure 1 is a fragment `Activity` and its menu items are text `Views`. These `Views` are defined in the main menu layout XML file and this file is referenced in the menu fragment `Activity` by calling the setContentView() function.

`Views` are responsible for event handling. `Input events` may be button clicks, edit text, touch, etc. To respond to an event of a particular type, the `View` (or `GUI element`) must register an appropriate event listener and implement the corresponding callback method (called by the Android Framework when the `View` is triggered by user interaction). For example, if a button is to respond to a click event it must register to View.onClickListener event listener and implement the corresponding onClick() callback method. When a button click event is detected, the Android framework will call the onClick() method of that particular `View`.

An `event sequence` is an ordered set of input events. The term `state` in this paper refers to GUI state which is a collection of GUI information about the current screen and all the GUI elements in it. Amaze file manager App shown in Figure 1 has three different states although it remains in the same `Activity`. We refer to change-affected GUI elements as `target GUI elements`, Activities containing a target GUI element as `target Activities` and states containing a target GUI element as `target states`.

As Android Apps are event-driven, inputs are normally in the form of events. Writing or recording input events manually can be arduous and time-consuming [17]. Automated input event generation to test Android Apps is an active area of research. A summary of existing research in Android testing is presented in the next section.

IV. Related Work

CAT is the first Android GUI testing work focusing on App updates. In this Section, we summarise existing work on Android GUI test generation, split into random and model driven testing. We also discuss related research in regression test selection that selects tests based on App updates.

Random Android GUI Testing. Android Monkey [18] is a popular random testing tool that examines the GUI and randomly selects events to be exercised in the current state until the number of exercised events exceeds the limit set by user. DynoDroid [19] uses heuristics to select input events rather than being fully random. However, DynoDroid has not been maintained for years and only supports Android version 2.3.5 (Android 10 is current version). Wetzlmaier et al. [20] amplify existing test inputs by injecting random test inputs. This technique gives the user more control than Monkey. None of the existing random testing tools focus on App updates.

Model-based Android Testing. DroidBot (DB) [16] and DroidMate (DM) [21], [2] focus on generating test inputs based on GUI models. DM guides test input generation on-the-fly using the GUI model. DB queries the GUI model of the subject App, computes and executes possible events in this model. DB also provides an easy to use interface for App exploration. CAT leverages this feature in DB for depth-first App exploration from the start `Activity`.

Different from static GUI model-based test generation, Ape [22] dynamically optimises the GUI model by leveraging the runtime information during testing. During App exploration, Ape uses a decision tree-based representation and continuously refines the GUI model with the aim of maintaining a good balance between the model size and model precision.

a) Regression test selection: Several studies have examined selection of regression tests based on App updates and their impact. Focus of CAT is different - input generation for change affected elements. None of the regression test selection work perform input generation for changes. Nevertheless, both CAT and regression test selection techniques rely on change affected elements identified using change impact analysis. We summarise change impact analysis in Android regression test selection below.

Redroid [13], [15] and ReTestDroid [12] are regression test selection techniques that compare Java source files from original and updated App versions to identify changes and compute change impact at the source code level. Regression tests that exercise change impacted code are selected by the tools. ReTestDroid handles more Java features than Redroid, such as fragments, native code and asynchronous tasks. Both tools perform change impact analysis at the source code level, not considering GUI elements and are used for test selection but generation. CAT performs change impact at the source code and GUI level and focus on test generation for change impacts.

QADroid [11] and ATOM [14] also perform test selection for regression versions of Apps. QADroid analyses impact of App updates on code and GUI elements. QADroid, like CAT, builds call graphs based on FlowDroid [23] and links events to function calls using event-function bindings defined in source code. QADroid does not support change impact analysis for dynamic GUI elements, as it does not support Java reflection. ATOM [14] builds an event-flow graph for each App version, whose nodes are `Activities` and edges are events that cause `Activity` transitions. It then computes a delta graph using event-flow graphs of the updated and original App versions. Only events existing in the delta graph are picked for regression test selection.

V. Our Approach

We present CAT – Change-focused Android GUI Testing – framework that provides, 1. Change impact analysis and 2. Test input generation for change impacted GUI elements in Android Apps. Our framework is publicly available at https://github.com/CATAndroidTesting/CAT.

The workflow of CAT is presented in Figure 2. The input files to CAT are an APK file and a user-provided change-set for the App version under test in a JSON file. The JSON file lists the signatures of updated classes and functions in the source code. Output is a set of GUI event sequences to exercise the changes and change impact in the App. Steps in CAT's workflow are as follows,

1. Input Preprocessing. The APK file is first analysed to produce a list of layouts in the App, Androidmanifest.XML file, and a call graph representing calling relations between functions in the Java code.

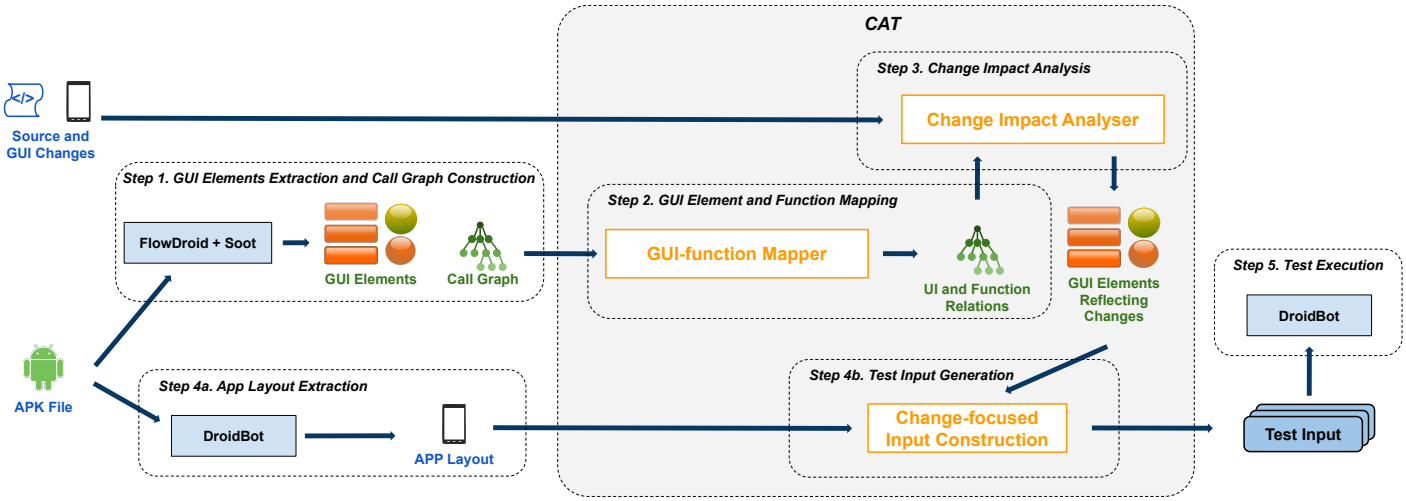

Fig. 2: Workflow of the CAT framework.

2. **GUI Element and Function Mapping.** CAT then traces GUI elements to the underlying listener functions in the source code. Using this tracing information and the call graph from Step 1, CAT generates a combined graph for tracing between source code functions and GUI elements.
3. **Change Impact Analysis.** For functions in the source code that are marked as changed, CAT analyses impact of the changes at the GUI level by tracing paths from the changed functions to GUI elements in the combined graph, generated by Step 2 and produces a list of target GUI elements that are affected by the App update.
4. **Test Input Generation.** For target GUI elements from Step 3, CAT generates test inputs as GUI event sequences that interact with them at least once.
5. **Input Execution.** We execute GUI event sequences generated by CAT on a test execution engine built on top of DB [16].

We discuss details of the design and implementation of these working phases of CAT in the rest of this section.

A. Step 1: Input Preprocessing

We use FlowDroid [23] along with Soot [24] to preprocess APK files to generate the following - (1) a list of layout files, (2) AndroidManifest.XML file, and (3) a call graph whose nodes are functions in the source code and edges represent calling relations.

Customisation: In its current form, FlowDroid does not expose `layout` files embedded within other `layout` files and consequently the GUI elements defined in them. CAT relies on mapping all GUI elements to `Activities` and functions in the source code for subsequent steps that analyse change impact and generate GUI event sequences. We, therefore, augment FlowDroid with a data collector that allows us to gather information on embedded layout files.

B. Step 2: GUI Element and Function Mapping

In this step, we build a mapping from functions in Java code to GUI elements and `Activities` using the artifacts generated by FlowDroid in Step 1. This mapping will be useful in determining the GUI events that will help exercise the changed functions in Java code. The mapping is built in two stages - (1) Mapping functions in Java to GUI elements, and (2) Mapping GUI elements to the `Activity` class.

For the first stage, we initially take the call graph produced by FlowDroid and extract the underlying undirected graph from it that captures function dependencies. We refer to this undirected call graph as function graph. Next, we identify `listeners` in the function graph that get triggered when there is an interaction with a corresponding item in the GUI. We then expand the function graph with additional nodes for GUI elements and edges between listener nodes and the GUI element nodes they register an event for. Output of the first stage is the expanded function graph that contains functions and GUI elements as nodes with undirected edges representing calling relationships.

In the second stage, we start by extracting all `Activities` from AndroidManifest.xml. For each `Activity`, we track the `setContentView()` method that is used to render the associated layout. We also track the `inflate()` method, if present, that is used to change the layout after the `Activity` starts. We use these methods to map each `Activity` to the layout it is associated with. The layout file lists all the GUI elements that will appear to the user for that `Activity`. We use the GUI element listing in the layout file along with the `Activity - Layout` mapping to build an association between GUI elements and the `Activity` they reside in.

The information from the first and second stages can be merged using GUI elements as the key values connecting both. Merged information allows us to trace functions in Java to GUI elements that can trigger them and further to `Activities` where the user can interact with these GUI elements. We refer to this merged information as *combined GUI-function map*

Figure 3 shows the utility of the combined GUI-function map, built from the Amaze file manager example, to trace from a changed function in the source code to a GUI element and then to an `Activity` that the GUI elelment is contained

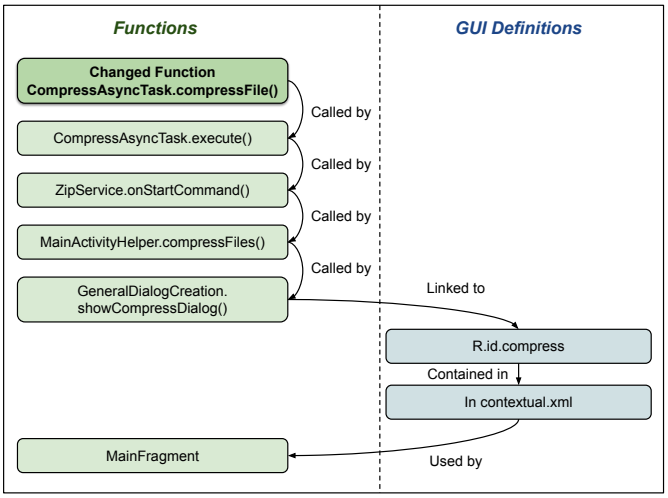

Fig. 3: Tracing impact of changed function to GUI element using combined GUI-function map.

in. Dependencies between functions are gathered from the function graph.

Dynamic GUI elements are commonly used in Android development with Java reflection. A dynamic GUI element can be referred to in the source code with a symbol whose value is resolved at runtime. To support change impact involving dynamic GUI elements, we annotate these locations so we know to explore them at runtime. When the changed `Activity` is reached, we record which GUI element leads to this `Activity`. It is worth noting that existing static change impact analysis tools, such as QADroid [11], cannot precisely handle these dynamic features.

C. Step 3: Change Impact Analysis

This step starts from the JSON file with signatures of changed functions (*packageName.className.functionName(parameterList)*) identified by the developer. We then perform a depth-first traversal of the combined GUI function map starting from the changed functions. We are only interested in visited nodes that are GUI elements for test input generation in the next step. Transitive closure of all such visited nodes gives the set of target GUI elements. Events that interact with the target GUI elements are capable of executing the changed function in the source code. As shown in Figure 3, interacting with the widget with the id `compress` in the `MainFragment Activity` can trigger the changed function `compressFiles()` through a chain of internal function calls. Output of this step is the set of target GUI elements.

D. Step 4: Test Input Generation

Test input generation with CAT is built on top of DB [16]. CAT uses DB's depth first exploration from the start `Activity` to examine different states, checking if the target state (screen with target GUI element) is entered. Once target state is entered, CAT generates events prioritising interactions with target GUI elements in this state. For increased rigor, CAT generates length 3 event sequences, rather than a single

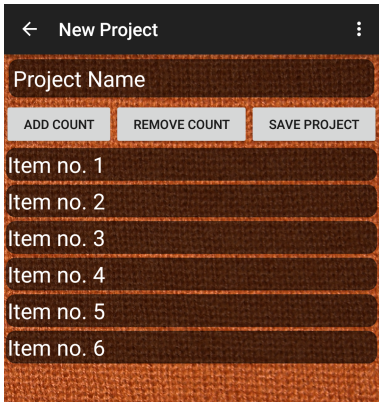

Fig. 4: `New Project` screen in the `BeeCount` App

event, to interact with the target GUI element. These steps in input generation are discussed in more detail below.

a) Check for the target state: CAT monitors the states entered during GUI exploration, checking if each new state contains a target GUI element identified in Step 3. Once the target state is entered it moves to the next step of generating event sequences that prioritises interactions with the target elements.

b) Generating event sequences: For thorough testing of the target element, it would be desirable to interact with it in different contexts, where a context is defined by the sequence of events leading to it. To balance rigor and feasibility, Memon et al. [25] use length-3 event sequences in their GUI testing work. Inspired by this, CAT generates length-3 event sequences for interacting with the target element. CAT is the first Android testing tool to consider permutation of events leading to a GUI element interaction.

We illustrate length-3 event sequences generated by CAT for the `New Project` screen in the `BeeCount` App, shown in Figure 4. Change impact analysis in Step 3 marks the `SAVE PROJECT` as the target GUI element impacted by changes. CAT builds length-3 event sequences that click the `SAVE PROJECT` option at least once in the sequence. Example sequence generated by CAT is click `ADD COUNT` – `REMOVE COUNT` – `SAVE PROJECT`. Sequences like this enable checking the behaviour of `SAVE PROJECT` using different combinations of prior events.

It is worth noting that some generated events can cause the app to leave the target state. When this happens, CAT uses the recorded sequence of input events to re-enter the target state.

E. Step 5: Input Execution

Event sequences generated by CAT can be executed on the Android emulator and real devices. For some event sequences, execution of an event may leave the target state, as mentioned earlier. The remaining events in the sequence can only be executed when the App goes back to the target state. To enable this, CAT gives unique IDs to events and event sequences and marks the execution status of each sequence. When a sequence has remaining events to be performed but the emulator leaves the target state, the paused events are added to a queue. CAT then uses a pre-recorded event sequence to go back to the target state and execute remaining events in the queue.

VI. Experiment

We evaluate feasibility and effectiveness of CAT in generating GUI events that exercise target GUI elements. We use the following Apps in our evaluation,

Open source - We use 21 Android applications from the F-Droid App market [2] that has a catalogue of free and open source Android applications. We start with open source projects that allow us to check the code diff and commit comments to detect and mark code changes which are then input to CAT. We selected top rated apps in F-Droid with a well documented commit history across multiple versions. Table I lists the names, versions and change information for the Android Apps used in our experiment. For each of the open source App versions, we manually collect changes in the App by reading commit comments for the APK file release. A JSON file with signatures of changed functions is then input to CAT for change impact analysis.

Commercial - We use two popular commercial apps, `TikTok` and `Huoshan`, #22 and 23 in Table I (in bold), used widely for sharing videos and social networking. It is worth noting that we only had access to the APK files for these two apps. Information on changes in the App versions were provided by developers at Bytedance Co. as we did not have access to the source code. According to the developers, changes in the `TikTok` and `Huoshan` versions affect over $1.5K$ lines of code in each.

We investigate the following research questions:

Q1. Change Impact Analysis: *Is CAT able to perform change impact analysis from changed functions in code to GUI elements?*
To answer this question, we use CAT to first build the combined GUI-function map that contains both the function graph and GUI-function mapping for all Apps in our dataset. We then traverse the combined map from the changed functions to GUI level to identify target elements. We manually verify if the GUI elements identified are correct and complete by going through the source code for each subject App. For the commercial Apps, the developers verified the GUI elements identified by CAT.

Q2. Test Input Generation for Changes: *Can CAT generate GUI events to exercise target GUI elements faster and more rigorously than popular testing tools – DroidBot (DB), DroidMate-2 (DM)?*
For each open source Android application, we run CAT, DB, DM to generate 1000 test inputs, where each test input is a GUI event[3]. We generate 2000 test inputs for each of the commercial Apps, `TikTok` and `Huoshan`, as they are larger and more complex. `TikTok` and `Huoshan` are 101 and 51 MB in size, respectively, while all the open source Apps are less than 20 MB. We compare how quickly the tools start interacting with the target GUI element. We measure rigour in exercising target GUI elements as number of generated events that interact with the target elements.

Q3. Bug Finding: *Is CAT able to detect previously undetected bugs in our App dataset?* In this question, we assess if CAT is able to identify previously undetected bugs in our dataset of 23 Android Apps. We compare CAT against DB and DM in this assessment.

Selected Tools. We select test input generation tools DB and DM for our comparison since these were reported in the literature as being easy-to-use and providing high function coverage. Monkey [18], a popular random testing tool, and Ape [22] are not used in our comparison as they only report screen coordinates of GUI interactions, and do not provide details of the interacted GUI element. Mapping the coordinate information to target GUI elements is not trivial, making comparison with our tool difficult.

We did, however, request developers of `TikTok` and `Huoshan` to compare effectiveness of CAT against their in-house GUI exploration tool which is similar to Ape in its approach. The in-house tool does not gather information on interacted GUI elements, making direct comparison with CAT difficult. We did not have access to their in-house tool and therefore report developer observations on performance of CAT versus their in-house tool, where possible, in our results.

Our experiments for open source Apps are performed on an Android emulator running on Mac OS 10.15.6 with 16 GB memory and 2.6 GHz Quad-Core Intel Core i7 processor. The virtual device in the emulator is Google Pixel 2 with Android API 27. The two commercial Apps require more runtime memory, so for these we run the experiments on a device rather than an Android emulator. We use a Huawei P20 Pro device running Android 10. The dataset of 23 apps and scripts needed to replicate the experiments are available at https://github.com/CATAndroidTesting/CAT.

VII. Results

We present results from our experiment in the context of the research questions in Section VI.

A. Q1. Change Impact Analysis

CAT is able to analyse the impact of changes to GUI elements automatically for all the subject Apps. Change impact analysis could statically determine target GUI elements affected for 20 of the 23 Apps, which include the two commercial ones. The remaining three Apps, (`Tricky Tripper`, `Suntimes` and `Hibi`), allocate dynamic GUI elements that required CAT to perform runtime analysis to determine the target GUI elements (described in Section V-C). CAT is able to complete change impact analysis for all the open source Apps within 30 seconds and the two commercial Apps within 2 minutes. Changes in our dataset of Android Apps took different forms. We briefly discuss change impact analysis performed by CAT for these different change types.

1. New GUI element added. This is the most straight-forward scenario as the information on affected GUI elements is readily available. CAT locates the new GUI element in the layout XML file, marks the GUI element as target element. It then locates the `Activity` associated with the layout of the target

[2]https://f-droid.org/

[3]We fix number of events to be in line with DB and DM command line interface support.

TABLE I: Description of subject Apps

#	App Name	Version	Change Info
1	World Weather	1.2.5	Behaviour of setting personal API key
2	Amaze File Manager	3.4.3	Implementation of compressing files and folders
3	BeeCount	2.4.6	Function for saving projects
4	Diary	1.7.0	Behaviour of quitting an opened entry input window
5	Omni Notes	6.0.5	Behaviour of discarding modified notes
6	OpenTasks	1.2.2	Implementation of saving new projects
7	Simple Draw	6.1.0	Implementation of printing the picture
8	Simple File Maganer	6.7.3	Callback function for the creating shortcut button
9	Simple Solitare	3.13	Callback function for a checkbox in the settings window
10	WiFiAnalyzer	3.0.1	The About window is changed
11	Hibi	1.4.0	The Settings window is changed
12	Geological Timescale	0.4.1	Behaviour of changing App language is modified
13	DroidShows	7.11.1	Function for sorting list items
14	Suntimes	0.12.9	Callback function for the confirm button in the dialog window
15	Word Scribe	1.6.2	Callback function for the View Changelog button
16	Nani	0.3.0	Callback function for the Help button
17	Fate Sheets	1.2	Implementation of the function for sort list items
18	Lift	0.2	Callback function for saving and setting the fitness program
19	Currency	1.33	Callback function for the About button in the menu
20	Tricky Tripper	1.6.2	Implementation of the function for importing new projects
21	Open Money Box	3.4.1	Behaviour of changing App language
22	**TikTok**	14.03	The callback function for the Back button in the chat windows
23	**Huoshan**	10.07	The callback function for the badge widget in the living streaming window

GUI element, and marks it as target `Activity`. This scenario appears in 4 subject Apps namely `Simple Draw`, `World Scribe`, `Fate Sheets` and `Nani`.

2. Modification to existing `Activity`. The target GUI element in this case is the one that is able to enter the modified `Activity`. GUI elements implemented to render another `Activity` may be statically or dynamically allocated. 10 Apps in our dataset had this type of change, with 7 of them implementing static GUI elements to render the modified `Activity`, and the remaining 3 (`Tricky Tripper`, `Suntimes` and `Hibi`) with dynamically allocated GUI elements. CAT traces changes in the `Activity` source code to static target GUI elements for 7 Apps (`Currency`, `Geological Times`, `Wifi Analyzer`, `Simple Solitare`, `Diary`, `BeeCount` and `World Weather`). Dynamic target elements in the other 3 Apps are identified during depth-first App exploration.

3. Changes to Java functions. Starting from each changed function, CAT traverses the combined GUI-function map to retrieve the GUI element(s) and associated state and `Activity` impacted by the change. There were 9 Apps in our dataset with changes to functions and CAT was able to retrieve change affected GUI elements for all 9 Apps. Both commercial Apps in our experiment fall into this category. According to the developers, changes were made in the source code of callback functions that are linked to GUI elements. They confirmed that CAT correctly identifies the target GUI elements.

B. Q2. Test Input Generation for Changes

We assess and compare effectiveness of the tools in exercising the target GUI elements with respect to (1) how quickly they start interaction, and (2) number of target GUI element interactions. To account for non-determinism in the Android environment, we ran each tool 10 times for each App, generating 1000 GUI events for open source and 2000 for commercial Apps each time. Numbers reported in this section are averaged over the 10 runs.

1) Number of events to first target element interaction: Figure 5 shows the average number of events each tool used before it first interacted with the target GUI element for all 23 Apps. Smaller number of events is better, as it indicates the tool starts interacting with the target element faster. Failure labels on bars indicate the associated tool did not interact with any target GUI element.

a) Open source Apps: CAT is the fastest to start interacting with the target element, only needing 83 events, on average, versus 286 for DB and 258 for DM. For 18 of these 21 Apps in Figure 5, CAT needs fewer events than DB or DM to start target element interaction. This is because when target state is entered, CAT prioritises interaction with target elements unlike the other two tools. On average, DM uses 159 events to enter the target state and a further 99 elements to interact with the target GUI element. In contrast, depth-first exploration used by DB and CAT enables them to reach deeper screens faster. We remind the reader that CAT diverges from DB only after reaching the target state. Both tools use 81 events, on average, to reach target state. DB uses an additional 205 events to start interacting with the target element. On the other hand, CAT only needs 2 additional events to start target element interaction.

DM outperforms DB and CAT on two Apps: `BeeCount` (App #3) and `Trick Tripper` (App #20). For `BeeCount`, shown in Figure 4, the target GUI element is the `Save` button in the `New Project Activity`. Entering this `Activity` requires clicking the `New Project` button in the start screen. DM clicks this button earlier than DB and CAT (they click on a different button to first go into Settings, perform further events and then return to click the New Project Button). Similarly, For `Trick Tripper`, the target state containing the target GUI element can be entered after the first screen. DM clicks the button leading to the target state right away while DB and CAT explore many other events and `Activities` before entering the target state.

For `Simple File Manager` (App #8), all 3 tools failed

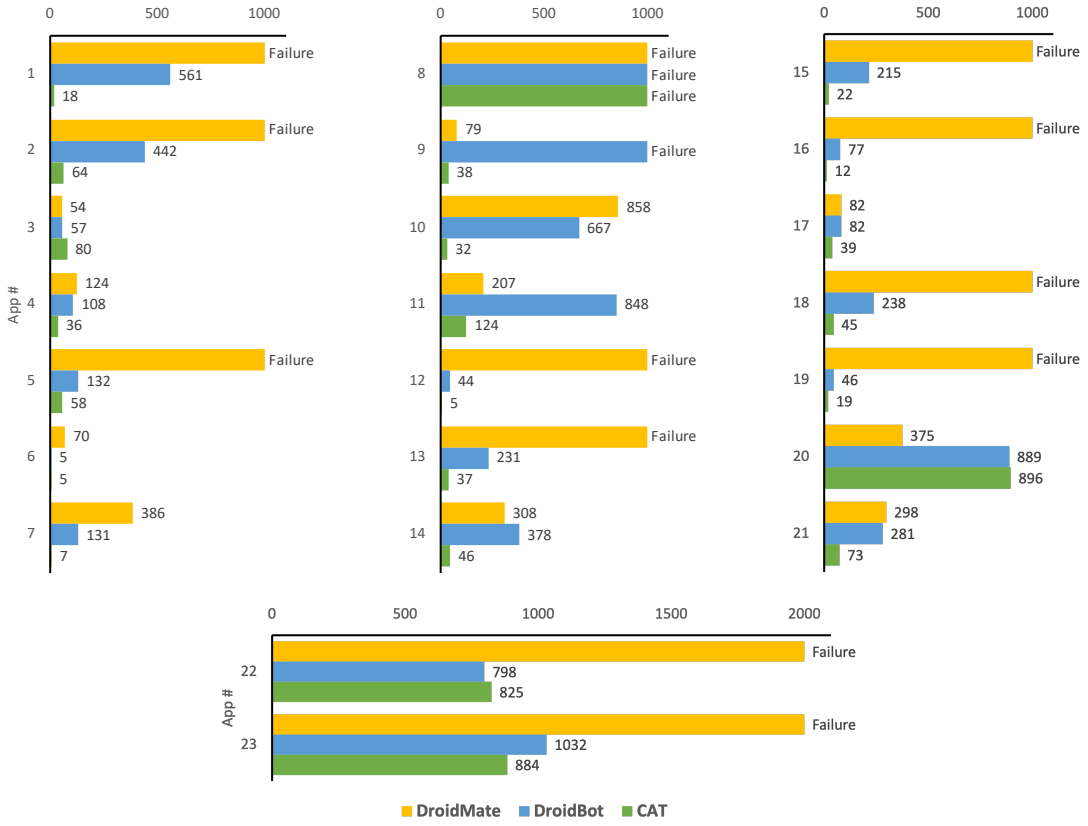

Fig. 5: Number of events taken to reach the first target GUI element interaction with DM, DB and CAT.

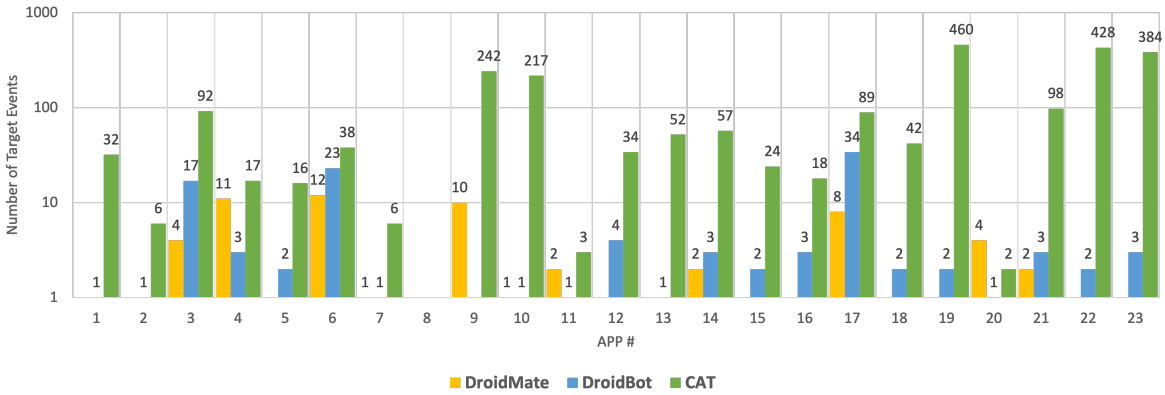

Fig. 6: Number of target GUI element interactions achieved by DM, DB and CAT

to interact with the target GUI element within 1000 events. Target GUI element for this App is a button for creating shortcuts. The state containing this button is not reached by all 3 tools as they get stuck on a screen for creating passwords, shown in Figure 7. To be able to leave this state, the tools had to provide input events that were exactly the same for initial and confirmation password. All three tools failed to achieve this within 1000 inputs. To avoid getting stuck in such non-target states, CAT provides the developer with an interface to specify an event sequence that leads it directly to the target state. When such a sequence is provided, CAT is able to interact with the target GUI element 90 times, with the first interaction happening after 17 events.

b) Commercial Apps: For `TikTok` and `Huoshan`, App# 22 and 23 in Figure 5, CAT and DB use similar number of events for first target element interaction – approximately 800 events for `TikTok`, 884 (CAT) - 1032 (DB) events for `Huoshan`. The commercial Apps, owing to their size and complexity, have a wider and deeper combined GUI-function map requiring a larger number of events to reach the target state. For `Huoshan`, DB uses 150 events more than CAT because after reaching the target state, DB does not interact

with the target element right away but instead randomly picks other GUI elements to interact with first. CAT, on the other hand, interacts with the target element immediately after the target state is reached.

DM completely fails to test the two commercial Apps. Log information indicates that it cannot retrieve GUI models when testing on a real device. Testing using the emulator was unsuccessful as the commercial Apps exceeded the memory capacity in the emulator.

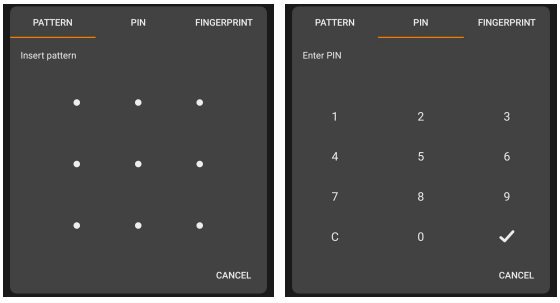

Fig. 7: Setting password in the Simple File Manager App

2) Number of target element interactions: Figure 6 gives the frequency of interactions with the target GUI element.

a) Open source Apps: We find DM and DB have limited number of target element interactions. DM performs 3 target element interactions on average for each App, and DB performs 5 interactions out of the generated 1000 events. CAT, on the other hand, performs 74 target element interactions that explore different event orders in length-3 sequences. This is because when CAT enters the target state, it stops exploring more states and provides a more rigorous event generation for target GUI elements with a higher likelihood of triggering change-related errors. As a result, CAT clearly outperforms DB and DM in number of target element interactions for all 21 open source Apps (as seen in Figure 6).

b) Commercial Apps: For `TikTok`, number of target GUI element interactions is 2 for DB versus 428 for CAT. For `Huoshan`, DB interacts 3 times versus 384 times with CAT. As DM has problems retrieving GUI models on real devices, it does not exercise target GUI elements in both the commercial Apps.

DB has dramatically fewer target element interactions than CAT because once the target activity is reached, DB does not prioritise interactions with the target GUI element. DB instead randomly picks one of the unvisited GUI elements on screen or navigates to the previous activity. `TikTok` and `Huoshan`, in comparison to open source Apps, have considerably more GUI elements in the target activity which reduces the likelihood of DB's random choice being the target GUI element.

In contrast, when CAT enters the target activity, it focuses solely on interacting with target GUI elements. Sometimes the target GUI element interaction may lead CAT to another activity, in that case CAT will navigate back to the target activity using the previously recorded event sequence. We also observed that CAT reliably interacted with the target GUI elements across multiple runs, with more than 300 interactions in each run.

In summary, we find CAT is able to exercise the target GUI elements more rigorously and reliably than DB and DM across both open source and commercial Apps.

c) Developer Observation: ByteDance's in-house testing tool does not provide information on GUI elements exercised. So in order to compare with CAT, developers examined App execution with input sequences from the different tools and found CAT interacted with the target GUI elements sooner and more frequently than their in-house tool. They also found CAT was more reliable than their in-house tool in time to and frequency of interactions over multiple runs.

ByteDance developers also found it easy to setup CAT and follow the command line interface of CAT to generate change impact results and run tests. After test execution, they can easily read experimental results from the CAT output.

d) Overhead: DM has the highest overhead in test input generation and execution. Average time taken by DM for each App, to generate and execute 1000 events, is 35 minutes while DB and CAT take 18 minutes. CAT introduces a negligible additional overhead of 5 seconds over DB. Higher overhead observed with DM is because DM consumes approximately 1 second more than DB or CAT after every event execution to extract the state model and generate the next event. The extra time accumulated across 1000 events results in significant difference.

For the commercial Apps, overhead incurred by DB and CAT is higher - approximately 0.5 seconds more for each GUI element interaction as they are executed on a real device and connection using USB debugging introduces some latency. CAT and DB took 1.5 hours to generate and execute 2000 GUI events for both commercial Apps.

C. Q3. Bug Findings

1) Open Source Apps: CAT uncovered change-related bugs in 2 of the 21 open source Apps – `World Weather` and `BeeCount`. DM did not reveal bugs in any of the Apps, while DB revealed a bug in the `World Weather` App but not `BeeCount`. Both `World Weather` and `BeeCount` App versions have changes at the source code level that impact GUI elements.

Latest version of `World Weather` changes the function to query weather information using a public API. CAT identifies the button for changing API key in the `Settings Activity` as the target GUI element. CAT's first interaction with the target GUI element results in a crash and the App stops execution. Both DB and CAT are able to crash the App by clicking the button for changing API key. However, once the App is restarted, DB does not produce additional events to interact with this target element as it proceeds to interact with other unvisited elements. CAT, on the other hand, continues to generate and execute events related to this button as it prioritises target element interaction. Interestingly, clicking this button only crashes the first time, further clicks do not result in crashes. This information about first-time only crash is useful for debugging and can only be provided by CAT as it interacts with the target element multiple times. The crash was reported to the App developer and is awaiting a fix at the time of writing this paper.

BeeCount App version in our dataset updates the listener function for Save button in the New Project activity. CAT identifies the New Project activity as the target state and the Save button as the target GUI element. The following event sequence generated by CAT – Edit Text, Click Menu, Click Save – reveals a bug in the App. Clicking the Menu button after editing text in the New Project activity loses the edited text. As a result, clicking the Save button does not perform the expected action of saving the edited text. We manually inspect executions of the event sequences to detect unexpected behaviour. The App is expected to save the text when leaving the New Project Activity so that when the user later returns to this Activity the previously entered text is retained. This bug in the BeeCount App is not revealed by DM and DB. Both DM and DB interact with the target Save button. However, they are unable to generate the event sequence that triggers this bug - Clicking Menu button after editing text, followed by clicking Save. Order of events is important for triggering this bug. CAT's focus on Length 3 event sequences interacting with the target element allows it to trigger bugs that are sensitive to event orders.

2) Commercial Apps: We found no bugs in both the commercial Apps. This is not surprising as we use release versions of TikTok and Huoshan that were extensively tested before their release. To assess the effectiveness of CAT and other tools in detecting change-related bugs in a commercial App, the App developers manually seeded a runtime exception within change affected source code in TikTok. The seeded bug is expected to crash the App when executed. The App developers generated input GUI events using their in-house testing tool, CAT and DB for this faulty TikTok version. It is worth noting that we were not involved in seeding the bug or running the input generation tools on the faulty version.

a) Developer Observation: The developers found CAT reliably detected the seeded fault by causing TikTok to crash within 26 minutes. Repeating the experiment several times did not change this result. DB was unable to trigger a crash while their in-house tool took approximately 6 hours to trigger the seeded fault.

Developer-run experiments over the commercial Apps have led the developers to acknowledge that CAT is more effective in testing change-affected code in the two commercial Apps than their in-house testing tool with regards to number of target element interactions, time to first target element interaction and revealing a seeded bug of their own choosing. They also found CAT easy to use and commented that it worked seamlessly with their Apps without requiring any modifications.

D. Threats to Validity

A potential threat to internal validity is bugs in CAT's implementation. To mitigate this threat, we conducted careful code reviews and extensive testing. Further, the implementations are publicly available for other researchers and potential users to check the validity of our results.

Regarding the soundness of our approach, the change impact analysis results were checked manually by inspecting the source code of the subject Apps to check the correctness. We conducted several rounds of the manual inspection by different developers to mitigate the risk of manual mistakes or omissions.

A potential threat to the external validity is related to the fact that the set of Android Apps we have considered in this study may not be an accurate representation of the App under test. We attempt to reduce the selection bias by using a dataset of 21 open source Apps from different categories with a variety of Android features, and 2 widely used commercial Apps.

A threat to construct validity is caused by restricting the number of GUI events generated by all 3 tools to 1000 for open source Apps and 2000 for the two commercial Apps. Restriction to 1000 input events is used by DM and DB in their default settings and we used the same for CAT. We don't believe changing the number of input events will affect the relative performance of the tools as we expect all 3 tools to be uniformly impacted. A final threat to validity is the limited number of tools used in comparison. We used DB and DM as they are popular, well-maintained and easy to use. We attempted to include Monkey and Ape tools in our experiment but found they were difficult to compare with owing to the lack of information on GUI elements exercised. This threat is mitigated to some extent by comparing with ByteDance's in-house GUI testing tool that uses an approach similar to Ape, according to the developers.

VIII. Conclusion

We presented the CAT framework for GUI test input generation targeting Android App updates. CAT supports change impact analysis to identify GUI elements affected by updates. It then generates GUI event sequences for interacting with these target GUI elements.

We empirically evaluated CAT's performance by comparing it to DB and DM over 21 open source and 2 popular industrial Android Apps. We made the following observations in our experiment.

1) CAT is able to trace changes made at the source code level to affected GUI elements automatically.
2) For target states containing target GUI elements, CAT is able to generate length-3 event sequences.
3) CAT interacts with target elements sooner than DB and DM, requiring 83 events on average, versus 286 for DB and 258 for DM.
4) CAT interacts with target GUI elements more frequently than DB and DM – average of 74 interactions for CAT, 5 for DB and 3 for DM with open source Apps and 406, 3 and 0 interactions, respectively, for commercial Apps.
5) Change-related bugs are revealed by CAT in two Apps. Order of input events was crucial in revealing the bug on one of these Apps. Only CAT was able to reveal this event order sensitive bug owing to the length 3 event sequence used to interact with the target element. CAT was also able to reveal a developer seeded bug in one of the commercial Apps, TikTok, faster than the company's in-house tool. DB and DM failed to reveal this bug.

In sum, CAT outperforms DB and DM in testing App updates on all 21 open source Apps and the 2 commercial Apps. Additionally, for the commercial Apps, developers confirmed that CAT does better than their in-house testing tool in revealing seeded bugs and interacting with the target GUI elements.

REFERENCES

[1] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[2] N. P. Borges, J. Hotzkow, and A. Zeller, "Droidmate-2: a platform for android test generation," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 916–919.

[3] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[4] Y.-M. Baek and D.-H. Bae, "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*, 2016, pp. 238–249.

[5] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 377–386.

[6] T. Su, "Fsmdroid: guided gui testing of android apps," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 689–691.

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

[8] W. Choi, K. Sen, G. Necul, and W. Wang, "Detreduce: minimizing android gui test suites for regression testing," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 445–455.

[9] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 559–570.

[10] W. Song, X. Qian, and J. Huang, "Ehbdroid: Beyond gui testing for android applications," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 27–37.

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

[12] B. Jiang, Y. Wu, Y. Zhang, Z. Zhang, and W.-K. Chan, "Retestdroid: towards safer regression test selection for android application," in *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 235–244.

[13] Q. C. D. Do, G. Yang, M. Che, D. Hui, and J. Ridgeway, "Redroid: A regression test selection approach for android applications." in *SEKE*, 2016, pp. 486–491.

[14] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li, "Atom: Automatic maintenance of gui test scripts for evolving mobile applications," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 161–171.

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

[16] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[17] R. Sharma, "Quantitative analysis of automation and manual testing," *International journal of engineering and innovative technology*, vol. 4, no. 1, 2014.

[18] G. Developers, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, accessed: 2020-08-20.

[19] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[20] T. Wetzlmaier and R. Ramler, "Hybrid monkey testing: enhancing automated gui tests with random test generation," in *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, 2017, pp. 5–10.

[21] K. Jamrozik and A. Zeller, "Droidmate: a robust and extensible test generator for android," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 293–294.

[22] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[23] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[24] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[25] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing," in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 2001, pp. 256–267.