Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising

Yauhen Leanidavich Arnatovich*, Minh Ngoc Ngo†, Tan Hee Beng Kuan* and Charlie Soh*
*Nanyang Technological University
50 Nanyang Ave, Singapore 639798
Emails: {yauhen001, ibktan, csoh004}@ntu.edu.sg
†Singapore Institute of Technology
10 Dover Drive, Singapore 138683
Email: cristal Ngo@singaporetech.edu.sg

Abstract—Automated functional user interface (UI) testing of mobile apps is a challenging task due to the highly interactive nature of the app UIs, and thus it commonly fails to provide high code coverage. In this paper, we present DroidDEV, an automated UI exerciser which aims to achieve high code coverage for real-world Android apps. DroidDEV dynamically builds a finite UI flow graph, generates and injects UI-context-aware inputs to exercise all the widgets on all the UI screens through a best-first search path finding algorithm. We have evaluated DroidDEV on 20 real-world open-source Android apps and compared it with manual UI testing and two other prevalent automated approaches including MobiGUITAR and Monkey. On these subjects, DroidDEV reports code coverage similar to manual UI testing, and it outperforms MobiGUITAR and Monkey in terms of code coverage and exercising time.

Index Terms—Android, functional UI testing, dynamic analysis, UI flow graph, input generation, UI exerciser, best-first search.

I. INTRODUCTION

Android apps are becoming increasingly prevalent, yet due to the often pressing time to market, numerous apps are released without sufficient testing and auditing. Thus, there is a growing need for automated software quality assurance tools.

Android apps have non-trivial highly interactive UI-nature, therefore UI testing is an essential component of any Android app development life cycle and significantly contributes to the quality of the app. However, existing state-of-the-art automated UI testing tools for Android apps [1–7] are still insufficient to adequately exercise apps UI [5, 8] and thus cannot achieve high code coverage in automated UI testing [9]. In particular, existing automated UI testing approaches poorly exercise Android apps because they fail to address both of the following issues:

- Android apps have non-trivial UI models, which may have a large number of execution branches. As such, adequately exercising the apps functionality within a reasonable time is a challenging task.
- Android apps UI and functionality rely on sensible user inputs which are provided by humans. However, most of existing tools either generate inputs that are too trivial or require user to provide such inputs during app testing. As such, existing approaches may inherit drawbacks of manual UI-testing.

To overcome the limitations of existing UI testing approaches, in this paper, we propose a novel approach, namely DroidDEV (AnDroid Dynamic, Exerciser, high coVerage), for achieving high code coverage in mobile applications by exhaustively and systematically discovering all app UIs at any depth, and automatically generating UI-context-aware inputs to exercise all the widgets on all the discovered app UIs. To effectively and efficiently discover all the app UIs, DroidDEV dynamically builds a finite UI flow graph (∗-UFG) of the app to guide best-first search procedure.

We have successfully implemented DroidDEV in a tool for Android apps by expanding the UI Automator testing framework.1 The UI Automator testing framework provides a unique feature to dynamically dump the UI screen hierarchy of the foreground app UI into an XML file. The XML file is then used by DroidDEV to dynamically discover and exercise relevant widgets, and generate suitable inputs for different UI components.

In this paper, we have made the following contributions:

1) We proposed a novel approach, namely DroidDEV, to provide high code coverage in automated UI testing of Android apps through dynamic finite UI flow graph building, automated UI-context-aware input generation and best-first search UI discovery.
2) We successfully implemented DroidDEV in a practical tool for Android apps.
3) We conducted experiments on 20 real-world open-source Android applications to evaluate the performance of our approach in term of code coverage and exercising time with manual UI testing and two prevalent automated approaches including Monkey and MobiGUITAR.

II. PROPOSED APPROACH DROIDDEV

In this section, we provide detailed explanations of the proposed approach DroidDEV, focusing on the algorithms it employs to perform fully automated exercising of Android apps UI to achieve high code coverage. We start by giving definitions in Section II-A. Section II-B provides an overview of the proposed approach. Sections II-C, II-D, and II-E describe

1developer.android.com/topic/libraries/testing-support-library
the three main algorithms used by DroidDEV including SensINJECT, DynaEXE and BestPath for automated input generation, dynamic execution and best-first search path finding algorithms, respectively.

A. Definitions

For further explanations of our proposed approach, in this section, we provide definitions which are used throughout the paper. Note that each UI screen might have one or more UI widgets, each could be editable (e.g., “EditText”) or non-editable (e.g., “Button”). A non-editable UI widget allows transition to the same or another app UI. Within the context of this paper, we consider that two UIs are equivalent if the number of UI widgets and their attributes “class” is the corresponding XML hierarchy views match. The UI widget attributes “text” (only for editable widgets), “NAF”, “bounds”, “focused”, “enabled”, “checkable”, “scrollable”, “long-clickable”, “selected”, “focused”, “clickable”, “checked” are excluded from the comparison since these attributes do not imply two UIs are not equivalent.

Definition 1: UI Flow Graph (UFG) is defined as a triple \( g = (U, W, L) \), where \( U \) is the set of non-equivalent UI screens, \( W \) is the set of UI widgets, \( L \) is the set of links where each link “\( i \)” is a directed edge from a UI widget “\( w \)” to a UI screen “\( u \)”, denoted as \( l = (w, u) \).

Definition 2: Finite UFG (\( f\)-UFG) is an UFG \( g \) that does not contain any sequence of links \( l_m, l_{m+1}, \ldots, l_k \), where \( l_m = (w_m, w_{m+1}), l_{m+1} = (w_{m+1}, w_{m+2}), \ldots, l_k = (w_k, w_m) \) and \( l_n \in L, w_n \in W, w_n \) on \( u_n, u_n \in U \), where \( n = m, m + 1, \ldots, k \) and \( m > 0, k > 0, m < k \). This means that a \( f\)-UFG is an UFG that does not contain any sequence of links which forms a “loop” between app UIs. Figure 1 provides an example of an UFG of an Android app with five UI screens. In Figure 1, \( u_i \) represents an UI screen, \( e_{ij} \) represents an editable widget, and \( w_{ij} \) represents a non-editable widget, where “\( i \)” is the index of the UI screen, and “\( j \)” is the index of the widget on the app UI screen. On each app UI, \( w_{ij} \) allows transition to the same or another app UI as indicated by the arrows. Note that the UFG in Figure 1 is not finite. This is because the following sequence of links \( l_1 = (w_{12}, u_2), l_2 = (w_{21}, u_3), l_3 = (w_{32}, u_4), l_4 = (w_{41}, u_1) \) forms a “loop” in the UFG.

B. DroidDEV: Overview of approach

The overall workflow of DroidDEV is given in Algorithm 1. Initially, the set of covered UIs \( \text{coveredUIs} \) (line 2) and \( f\)-UFG (line 3) are set to empty. DroidDEV runs the app (line 4) to render the first app UI screen (line 5). Next, the \( f\)-UFG of the app is dynamically built (lines 6–35) as DroidDEV exercises the app until all the widgets on all the discovered app UIs have been exercised. At every iteration of the exercising cycle, DroidDEV adds the current UI screen with its widgets into the \( f\)-UFG if it is not in the \( f\)-UFG (lines 7–9). UI-context-aware inputs will be automatically injected into all the editable widgets using the DroidDEV’s SensINJECT algorithm (lines 10–12). Afterwards, a non-editable unexercised relevant widget \( nWid \) of the current UI will be retrieved (line 13) and automatically exercised by DroidDEV’s dynamic execution engine, namely

![Figure 1. Example UFG of Android app.](image-url)
DynaEXE works by generating the relevant UI-events to exercise the non-editable widget \( nWid \), depending on the type of the widget. For comprehensive app exercising, DynaEXE supports most common types of UI-events. The set of UI-events that are currently supported by DroidDEV is given in Table I. Exercising the \( nWid \) renders a subsequent UI screen. Afterwards, DynaEXE adds the link from the \( nWid \) to the subsequent UI screen in the \( f-UFG \) if the subsequent UI screen is not in the \( f-UFG \). As such, DynaEXE ensures that the \( f-UFG \) does not have any loop. During the execution, DynaEXE monitors the execution flow. If there is any disruption in the execution flow, i.e., a widget transition that leads to a loop in the \( f-UFG \), DynaEXE returns FALSE as the result of the execution to indicate a discontinuity in the execution flow (line 14), otherwise returns TRUE. After DynaEXE execution, DroidDEV obtains the lastly discovered app UI from the \( f-UFG \) (line 15). The lastly discovered UI is the UI that was lastly added into the \( f-UFG \). DroidDEV checks if the lastly discovered UI is fully exercised (i.e., all its widgets are exercised) (line 16). If so, DroidDEV adds the lastly discovered UI into covered UIs (line 17) and removes it from the \( f-UFG \) (line 18). Afterwards, the lastly discovered app UI in the \( f-UFG \) is changed.

In line 20, DroidDEV updates the current UI after DynaEXE execution. If the current UI screen on the testing device is not in the \( f-UFG \) but has been fully exercised (line 21), DroidDEV will not proceed with the current UI. Instead, DroidDEV generates a system key event “Back” (line 22). This event returns to any previously discovered app UI. Thus, the current UI is updated (line 23). If the current UI is not in the \( f-UFG \) (line 24), DroidDEV continues the exercising procedure (line 25). Note that, depending on the app logic, system key event “Back” may exit the app. Such case is handled by DroidDEV by restarting the app and continuing the exercising procedure.

If DynaEXE does not indicate a continuity in the execution flow (line 28), DroidDEV finds a path to the lastly discovered UI in the \( f-UFG \) (line 29) and retraces the path to reach the lastly discovered UI (line 30). The retracing procedure (line 30) goes through all the widgets on the path \( p \), generating inputs using SensINJECT for editable widgets and exercising non-editable widgets along the path to reach the lastly discovered UI in the \( f-UFG \). This is an important part of DroidDEV which ensures that if there is any discontinuity in the exercising procedure, DroidDEV can track back the lastly discovered UI and exercise its yet unexercised widgets. DroidDEV automatically terminates the exercising process once the \( f-UFG \) is empty or the set of covered UIs contains the set of UIs from the \( f-UFG \) (lines 32–34).

In this workflow, DroidDEV employs the following three main techniques:

1) SensINJECT (Sensible UI-context-aware input INJECTION) automatically injects UI-context-aware inputs to the foreground app UI which is currently displaying on the Android device screen.

2) DynaEXE (Dynamic EXErciser) automatically exercises a non-editable UI widget by generating the relevant UI-event at a time. DynaEXE ensures that the \( f-UFG \) does not have any loop.

3) BestPath (Best-first search Path finding algorithm) performs a UI search in order to automatically build a path leading to the lastly discovered UI which was lastly added into the \( f-UFG \).

In the following sub-sections, we explain each of the main techniques in details.

### C. SensINJECT: UI-context-aware input generation

DroidDEV automatically injects UI-context-aware inputs to editable UI widgets by performing a context analysis of a given editable UI widget. In Algorithm 2, we provide details of the input injection process for an editable UI widget \( eWid \), namely SensINJECT. SensINJECT extracts relevant input types from three UI widget attributes including “text”, “resource-id” and “content-desc”. SensINJECT identifies the relevant input type of the editable UI widget (line 2) by referring to a dictionary called SensDIC, which essentially consists of the commonly used input types and their corresponding input values. We describe SensDIC in more details in the following sub-sections. Based on the input type, SensINJECT searches for the input value in the SensDIC (line 4) and injects it into the relevant editable widget (line 5).

As SensDIC does not consist of all possible input types, there might be cases where SensINJECT could not extract any

<table>
<thead>
<tr>
<th>Event type</th>
<th>Action(s)</th>
<th>Hardware key(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Touch</td>
<td>Click, LongClick</td>
<td></td>
</tr>
<tr>
<td>Motion</td>
<td>Swipe, Scroll</td>
<td></td>
</tr>
<tr>
<td>Trackball</td>
<td>Roll, Press</td>
<td></td>
</tr>
<tr>
<td>Keypress</td>
<td>Input Injection</td>
<td></td>
</tr>
<tr>
<td>“Major” navigation</td>
<td>“Menu”</td>
<td></td>
</tr>
<tr>
<td>“System”</td>
<td>“Home”, “Back”</td>
<td></td>
</tr>
</tbody>
</table>

### Table I: TYPES of EVENTS SUPPORTED BY DROIDDEV

<table>
<thead>
<tr>
<th>Event type</th>
<th>Action(s)</th>
<th>Hardware key(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Touch</td>
<td>Click, LongClick</td>
<td></td>
</tr>
<tr>
<td>Motion</td>
<td>Swipe, Scroll</td>
<td></td>
</tr>
<tr>
<td>Trackball</td>
<td>Roll, Press</td>
<td></td>
</tr>
<tr>
<td>Keypress</td>
<td>Input Injection</td>
<td></td>
</tr>
<tr>
<td>“Major” navigation</td>
<td>“Menu”</td>
<td></td>
</tr>
<tr>
<td>“System”</td>
<td>“Home”, “Back”</td>
<td></td>
</tr>
</tbody>
</table>

### Algorithm 2: SensINJECT

Input: \( eWid \) - editable UI widget

1. \( \text{procedure} \) SensINJECT(\( eWid \))

   \[ \begin{align*}
   \text{inputType} &= \text{SensDIC.GetInputType}(eWid); \\
   \text{if} (\text{inputType} \neq \varnothing) \text{ then} \\
   \text{inputValue} &= \text{SensDIC.GetInputValue}(\text{inputType}); \\
   \text{Inject}(\text{inputValue}, eWid); \\
   \text{else} \\
   \text{defaultValue} &= eWid.GetDefaultValue(); \\
   \text{if} (\text{defaultValue} \neq \varnothing) \text{ then} \\
   \text{Save}(\text{defaultValue}, eWid); \\
   \text{Inject}(\text{defaultValue}, eWid); \\
   \text{else} \\
   \text{result} &= \text{InjectRandomNumericValue}(eWid); \\
   \text{if} (\text{result} \neq \text{FALSE}) \text{ then} \\
   \text{InjectRandomStringValue}(eWid); \\
   \end{align*} \]
input type from the widget attributes. In this case, SensINJECT applies an input generation procedure (lines 6–17) as follows:

- If the editable accessible widget contains a default value (i.e., an actual value or hint) which is set by the app itself, SensINJECT saves such a default value for the future input generation (line 9), and injects the value into the widget (line 10).
- If the editable accessible widget does not contain any default value, SensINJECT generates a random input as follows: (1) SensINJECT tries to inject a random numeric integer value within the [0, 100] range (line 12); (2) otherwise, if the numeric value is rejected by the app (e.g., only string values are allowed), SensINJECT injects a random string value (line 14).

1) How SensDIC was built?: We build SensDIC as one part of the proposed approach for the input injection. SensDIC consists of commonly appeared editable widget input types such as email address, password, login, server address, server port, URL, and phone.

To build SensDIC, we manually inspected 100 selected apps from Google Play, which require complex input such as Skype, Facebook and Email, and others. Using the XML dumps produced by the UI Automator testing framework, we selected all editable accessible UI widgets, and extracted text terms of their specific UI attributes such as “text”, “resource-id”, and “content-desc”. Since they are created by programmers, their text terms usually reflect well the actual meaning of the required input types, although the quality of the text terms somewhat varies depending on the developers. Thus, we exploit the text terms of the UI attributes to compile our own SensDIC with the selected input types such as email address, password, login, server address, server port, URL, and phone.

It is worth emphasizing that SensDIC consists of input types with the corresponding input values. For example, for an editable UI widget which requires the email input type, SensDIC defines an input value “email@example.com”.

Optionally, for the input types in SensDIC, the realistic textual inputs can be provided by the user in the config.properties file so that they will be injected during app exercising instead of the input values in SensDIC.

2) Why SensDIC was built?: SensDIC was built to overcome the limitations of existing automated input generation approaches [1] where inputs are randomly generated and they are trivial for most of Android app UIs which are specifically designed for human interaction. For example, if an app requires a login and password, it is hard for an automated tool to generate sensible values for these kinds of inputs, which are normally provided by a human user. As SensDIC contains commonly used input types, SensINJECT is context-aware of the input type of a particular editable UI widget and thus, is able to search and inject a suitable input value into the widget if the input type of the widget is in SensDIC.

D. DynaEXE: Dynamic exerciser

In Algorithm 3, we show the procedure for DynaEXE. The input for DynaEXE is the current UI (currentUI) and a non-editable widget (nWid) on the UI. DynaEXE uses the XML files produces by the UI Automator framework to understand the structure of the current UI under exercising. DynaEXE first inspects the attributes of the nWid and generates a suitable UI event for the widget, and executes the event for the widget (line 2). By firing an UI event, the current UI screen might be changed. As such, the current UI should be updated (line 3). If the current UI is a newly discovered UI screen which is not in the f-UFG (line 4), the link from nWid to the current UI is added to the f-UFG (line 5). DynaEXE returns TRUE (line 6) to indicate that there is no disruption in the execution flow and DroidDEV can continue with the newly discovered UI. Otherwise, DynaEXE returns FALSE (line 8) to indicate that there is a disruption in the execution flow of the UI that requires DroidDEV to track back the last discovered UI in the f-UFG.

Algorithm 3: DynaEXE

Input: currentUI - app UI; nWid - non-editable UI widget
Output: continuity

1. procedure DynaEXE(currentUI, nWid)
2. GenerateRelevantUIEvent(nWid);
3. currentUI = GetCurrentUI();
4. if (currentUI \notin f-UFG) then
5. f-UFG.AddLink(nWid, currentUI);
6. return continuity = TRUE;
7. end
8. return continuity = FALSE;
9. end

E. BestPath: Best-first search path finding

The BestPath algorithm implements a best-first search algorithm [10] guided by a heuristic to find a path to the last discovered UI in the f-UFG. BestPath is shown in Algorithm 4. BestPath starts from the current UI. The heuristic that is used to guide the search is as follow. On UI, BestPath only selects a non-editable widget rWid that was most recently exercised and has links (line 2). The idea behind is that the chosen widget which satisfies the heuristic is likely to lead to the last discovered UI in the f-UFG. The UI (UI) and the chosen widget rWid is added to the path (line 3). For each outgoing link l from the selected widget rWid, BestPath recursively continues the search (lines 4–6). As each link l is a directed edge from a non-editable widget to a UI, GetEndUI function

Algorithm 4: BestPath

Input: UI - app UI, f-UFG - finite UFG
Output: path p to last discovered UI

1. procedure BestPath(UI, f-UFG)
2. rWid = GetRecentlyExercisedWidget(UI);
3. p.Add(UI, rWid);
4. foreach (outgoing link l from rWid) do
5. if (BestPath(l.GetEndUI(), f-UFG)) then
6. p = BestPath(l.GetEndUI(), f-UFG);
7. end
8. end
returns the UI where $rWid$ connects through the link. Note that the lastly discovered app UI does not have any outgoing links. Thus, when BestPath reaches the lastly discovered UI, it skips the loop (lines 4–6), and terminates by returning the built path $p$ (line 7).

III. DEMO OF DROIDDEV

Next, we illustrate the overall workflow of DroidDEV through a concrete example of an Android app the UFG of which is given in Figure 1. The app consists of 5 UI screens. DroidDEV exercises the app through the following steps:

- Run the app and obtain the first UI, i.e., $u_1$. Add $u_1$ and its widgets to the $f$-UFG. SensINJECT injects input into $e_{11}$. Afterwards, DynaEXE clicks the first unexercised non-editable widget $w_{12}$ so $u_2$ is rendered. Thus, DynaEXE adds the link $(w_{12}, u_2)$ and $u_2$ with its widgets to the $f$-UFG and it returns TRUE to indicate there is no disruption in the execution flow. The current UI is now $u_2$ and the execution flow continues.
- Repeat the previous step for $u_2$ and $u_3$. After exercising $w_{32}$, the current UI is $u_4$ and the execution flow continues.
- On $u_4$, DynaEXE clicks $w_{41}$, thus rendering $u_4$. As $u_4$ is already in the $f$-UFG, the edge $(w_{41}, u_4)$ will cause a loop in the $f$-UFG. Therefore, DynaEXE will not add the edge $(w_{41}, u_4)$ into the $f$-UFG. DynaEXE returns FALSE to indicate that there is a disruption in the execution flow (line 14, Algorithm 1, continuity equals to FALSE). DroidDEV updates the current UI to $u_4$ and BestPath finds the path to the lastly discovered UI, i.e., $u_4$. The path includes $(u_1, w_{12}, u_2, w_{21}, u_3, w_{32}, u_4)$. DroidDEV retracts this path. After retracing the path, the current UI is $u_4$.
- On $u_4$, DroidDEV retrieves the next unexercised, non-editable widget from $u_4$, i.e., $w_{42}$. DynaEXE clicks $w_{42}$, thus rendering $u_4$. Since $u_4$ is already in the $f$-UFG, the edge $(w_{42}, u_4)$ will not be added into the $f$-UFG. DynaEXE returns FALSE to indicate a disruption in the execution flow. After exercising $w_{42}$, $u_4$ has no unexercised widgets and thus $u_4$ is fully exercised. As such, DroidDEV adds $u_4$ with its widgets into the set of covered UIs and removes it from the $f$-UFG. As a result, in the $f$-UFG, DroidDEV changes the lastly discovered UI to $u_3$. DroidDEV also realizes that the current UI is $u_4$ which has been removed from the $f$-UFG and added into the set of covered UIs. Therefore, DroidDEV generates a system "Back" event (line 22, Algorithm 1). The "Back" event brings the application to any previously discovered UI. Without loss of generality, we assume that the event brings the application back to $u_1$. DroidDEV updates the current UI to $u_1$ and BestPath finds a path from $u_1$ to the lastly discovered UI in the $f$-UFG, i.e., $u_3$. The path includes $(u_1, w_{12}, u_2, w_{21}, u_3)$. DroidDEV retraces the path and updates the current UI to $u_3$.
- On $u_3$, DroidDEV retrieves the next unexercised, non-editable widget from $u_3$, i.e., $w_{33}$. DynaEXE clicks $w_{33}$, thus rendering $u_5$ which is not in the $f$-UFG. Thus, DroidDEV adds the link $(w_{33}, u_5)$ and $u_5$ with its widgets to the $f$-UFG, and returns TRUE to indicate that there is no disruption in the execution flow. DroidDEV updates the current UI to $u_5$ and the execution continues.
- On $u_5$, SensINJECT injects inputs into editable widgets $e_{51}$ and $e_{52}$. Afterwards, DynaEXE clicks the first unexercised non-editable widget $w_{53}$, thus rendering $u_4$. As $u_3$ is already in the $f$-UFG, the link $(w_{53}, u_3)$ is not added to the $f$-UFG. DynaEXE returns FALSE to indicate a disruption in the execution flow. After clicking $w_{53}$, all the widgets on $u_5$ are exercised. As such, $u_5$ with its widgets is added into the set of covered UIs and removed from the $f$-UFG. As a result, the lastly discovered UI in the $f$-UFG is changed to $u_3$. DroidDEV updates the current UI to $u_3$.
- Since all the widgets on $u_3$ have been exercised, DroidDEV adds $u_3$ with its widgets into the set of covered UIs and removes it from the $f$-UFG. As a result, in the $f$-UFG, DroidDEV changes the lastly discovered UI to $u_2$. Note that the current UI is $u_3$ which has been removed from the $f$-UFG and added into the set of covered UIs. As such, DroidDEV generates a system "Back" event to $u_1$. Afterwards, BestPath searches for a path from $u_1$ to the lastly discovered UI in the $f$-UFG, i.e., $u_2$. The path includes $(u_1, w_{12}, u_2)$. DroidDEV retraces the path and updates the current UI to $u_2$.
- The previous step is repeated for $u_2$ and $u_1$ and these two UIs with their widgets will be removed from the $f$-UFG and added into the set of covered UIs.
- At last, the $f$-UFG is empty and DroidDEV terminates its procedure. All the widgets on all the discovered UIs have been exercised.

IV. KEY QUALITY ATTRIBUTES OF DROIDDEV

DroidDEV is a novel automated UI testing approach which distinguishes itself from other approaches with the following quality attributes: Self-terminative, Automated, Effective and Efficient.

- **Self-terminative**: DynaEXE is designed to ensure that there is no loop in the $f$-UFG. As a results, DroidDEV is able to terminate automatically and thus it does not require the user to manually set a time limit or number of injected events to terminate the exercising process.
- **Automated**: SensINJECT extracts the data from the foreground UI of the app, which contains the information necessary to determine suitable UI-context-aware inputs with the help of SensDIC. To fully automate the UI testing process, DroidDEV ensures that input values are always provided.
- **Efficient**: In the $f$-UFG, BestPath employs the best-first search algorithm to build a path in order to reach the lastly discovered UI. Since DroidDEV always builds the $f$-UFG without loops, it guarantees that BestPath can always find such an app UI.
- **Effective**: DroidDEV revisits and exercises the lastly discovered app UI in the $f$-UFG until all its widgets
are exercised. This is done by using BestPath to build a path for the newly discovered UI and retracing the path to reach the UI. Note that exercising the widgets on the newly discovered app UI may discover new app UIs which are not in the f-UFG. In this case, the newly discovered app UIs will be added in the f-UFG and will be exercised. As a result, DroidDEV discovers all app UIs and ensures that all their widgets are exercised.

V. EMPIRICAL EVALUATION

In this section, we explain how we setup experiments to compare the performance of DroidDEV with MobiGUITAR [5], Monkey [11], and ManualTest (manual UI testing by a tester). MobiGUITAR is a tool for automated model-based testing of mobile apps. It implements both systematic exploration strategies (i.e., Depth-first and Breath-first) and random one. Monkey is a fully automated tool which generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. Monkey treats app under test as a black-box. It exercises an app by simply performing random actions at random positions on the UI.

A. Experimental setup

From F-Droid, we downloaded 20 real-word open-source Android apps. Note that most of code of these 20 Android apps is UI-related. Therefore, we ensure that the obtained code coverage correctly shows the ability of DroidDEV on exercising UI-related code rather than system-related code since system events are not supported. The first column in Table II gives the categories of the subject applications. The selected apps are across different domains. We selected more applications from the “Games” and “Multimedia” categories because these applications are generally highly interactive and challenging to test.

We ran DroidDEV, MobiGUITAR and Monkey for all the 20 apps on eight Android emulators, and averaged the obtained code coverage results. As DroidDEV is self-terminative, we do not limit its exercising time. However, for MobiGUITAR and Monkey, we need to set a time limit to terminate their exercising procedure since both tools are not self-terminative. Based on the recommended configuration, we keep 60 minutes exercising time limit for each app for MobiGUITAR. To fairly compare MobiGUITAR with Monkey, we set 60 minutes for each app for Monkey. Note that the settings for Monkey are different from MobiGUITAR. For Monkey, the number of events to be injected must be specified. Once the specified number of events has been injected, Monkey terminates the exercising process. In this experiment, we set the number of events to be 5,000,000 to enable Monkey continuously run for at least 60 minutes for each app. Once 60 minutes have elapsed, we kill the Monkey process.

To perform manual testing (i.e., ManualTest), we selected eight advanced Android users with knowledge in software testing on the Android platform. They all know the intricacies of Android app UI testing rather than common users. We assigned all the 20 apps to all the eight advanced users and averaged their code coverage results. To make a fair comparison with DroidDEV, we required the user to exercise each app until the user discovers all possible app UIs (according to his/her understanding about the app) in order to ensure that the maximal possible code coverage was achieved.

B. Code coverage analysis with EMMA

For code coverage generation, we use the EMMA tool. EMMA is an open-source tool for measuring and reporting Java code coverage and is included in the Android SDK by default. EMMA supports coverage types such as “class”, “method”, “basic block”, and “line”. Note that “line” coverage is not supported when testing app source code is not available. In our experiment, we use “basic block” as a coverage metric. According to EMMA, the “basic block” is a fundamental coverage metric, and other coverage metrics such as “branch”, “class”, “method”, and “line” can be derived from “basic block”.

C. Experimental results

In Table II, we show the “basic block” app code coverage obtained by DroidDEV, ManualTest, MobiGUITAR and Monkey (columns 3–6) and the exercising time taken by DroidDEV in the last column. From the last column in Table II, we can see that the average exercising time taken by DroidDEV for each app is equal to 15.76 minutes which is much shorter than the time limit (60 minutes for each app) for MobiGUITAR and Monkey. The exercising time for DroidDEV varies depending on the UI model of the app under test. The minimum time taken by DroidDEV is 0.35 minutes for the Core app, and maximum one is 117.82 minutes for the MemoPad app. The Core app is a game consisting of one single main UI requiring complex sequences of events leading to win or lose the game. In turn, the MemoPad takes much longer since it has more complex app UI model consisting of multiple distinct UIs with variety of widgets.

The coverage data given in Table II (columns 3–6) is computed as follows:

\[
Coverage = \frac{\text{Average number of covered basic blocks}}{\text{Total number of basic blocks}}
\]

As for ManualTest, the average number of covered basic blocks is computed by taking the average of the results given by the eight manual testers. As for DroidDEV, MobiGUITAR and Monkey, the average number of covered basic blocks is computed by taking the average of the coverage results given by the eight Android emulators.

1) DroidDEV vs. ManualTest: Table II shows that for most apps, DroidDEV performs similar to ManualTest. However, there are some noticeable differences in the code coverage of several apps, which are due to the specific nature of certain apps. For example, DroidDEV shows better performance for Ringtone Generator and Mytronome apps. We manually

\[\text{emma.sourceforge.net}\]
studied and found that these apps require more comprehensive exercising with long sequences of UI events, which DroidDEV, being an automated tool, can generate more efficiently.

For the apps such as Core, DivideAndConquer, and MunchLife, ManualTest shows better performance. It is simply explained by the fact that humans are better at providing short and more intelligent sequences of UI events than automated approaches. Furthermore, these three apps are all games and humans can better deduce the complex sequences of UI events required to win or lose the game. For the remaining apps, DroidDEV and ManualTest perform similarly. From Table II, we can see that DroidDEV covered 91.5%, and ManualTest covered 91.0% of each app code on average. Thus, DroidDEV is capable of achieving high code coverage.

2) DroidDEV vs. MobiGUITAR and Monkey: Table II also provides the coverage results for MobiGUITAR and Monkey. On average, DroidDEV outperforms MobiGUITAR and Monkey. It can be seen that the results obtained by Monkey vary from very low (4.8%) to very high (100%). Such code coverage results can be simply explained by the Monkey nature, i.e., Monkey generates random events at random positions on the app UI. Therefore, its code coverage results may significantly vary from app to app, and be affected by the density and physical size of the widgets. Thus, on average, Monkey shows lower coverage results compared to DroidDEV. In particular, DroidDEV covers 91.5%, while Monkey covers 71.9% of each app code on average.

Table II shows that, on average, MobiGUITAR provides low (44.9%) code coverage results. After our thorough manual investigation, we found that such results are due to the lack of support for complex UI events, e.g., motion and trackball UI-events, in MobiGUITAR. Instead, MobiGUITAR generates standard UI events such as “Click”, and “LongClick”, “Input Injection”, hardware keys, and system events such as “Home”, “Back”, and “Menu”. However, many selected apps such as AndroFish, DivideAndConquer, Battery Dog, Acrylic Paint, Coloring for Kids, and Dotty mainly rely on the complex user events such as motion (e.g., “Swipe” and “Scroll”), and trackball events (e.g., “Roll” and “Press”). As such, MobiGUITAR could not exercise the app functionality adequately. In summary, even with a much longer exercising time of 60 minutes for each app as compared to the time taken by DroidDEV (15.76 minutes on average), on average both MobiGUITAR and Monkey failed to achieve high code coverage.

VI. RELATED WORK

In this section, we provide an overview of randomized and state-of-the-art automated model-based testing approaches which implement systematic exercising of app UIs. Random testing [12, 13] is the most simple and primitive approach to exercising app UIs in a random manner, while model-based testing approaches [14, 15] use a UI-model derived from app under test either manually or dynamically during app exercising. Generally, model-based testing is the most suitable mechanism for guiding automated UI testing, and is usually combined with random testing, or dynamic symbolic execution.

Monkey [11] is a fully automated tool which generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. It can be used to stress-test applications in a random yet repeatable manner. Monkey simply performs random actions at random

---

**Table II**

<table>
<thead>
<tr>
<th>App category</th>
<th>App name</th>
<th>DroidDEV, %</th>
<th>ManualTest, %</th>
<th>MobiGUITAR, %</th>
<th>Monkey, %</th>
<th>DroidDEV, mins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>Dotty</td>
<td>100</td>
<td>100</td>
<td>55.9</td>
<td>100</td>
<td>1.21</td>
</tr>
<tr>
<td>Games</td>
<td>MunchLife</td>
<td>79.6</td>
<td>88.7</td>
<td>33.9</td>
<td>55.7</td>
<td>1.61</td>
</tr>
<tr>
<td>Games</td>
<td>AndroFish</td>
<td>95.2</td>
<td>91.9</td>
<td>4.0</td>
<td>42.8</td>
<td>10.87</td>
</tr>
<tr>
<td>Games</td>
<td>Core</td>
<td>85.1</td>
<td>92.9</td>
<td>53.2</td>
<td>88.2</td>
<td>0.35</td>
</tr>
<tr>
<td>Games</td>
<td>DivideAndConquer</td>
<td>86.2</td>
<td>92.5</td>
<td>41.5</td>
<td>78.4</td>
<td>4.69</td>
</tr>
<tr>
<td>Games</td>
<td>GM Dice</td>
<td>81.8</td>
<td>77.9</td>
<td>31.3</td>
<td>53.6</td>
<td>44.52</td>
</tr>
<tr>
<td>Graphics</td>
<td>Coloring for Kids</td>
<td>96.6</td>
<td>96.7</td>
<td>48.7</td>
<td>94.7</td>
<td>37.04</td>
</tr>
<tr>
<td>Graphics</td>
<td>Acrylic Paint</td>
<td>94.6</td>
<td>95.8</td>
<td>18.3</td>
<td>61.2</td>
<td>5.42</td>
</tr>
<tr>
<td>Multimedia</td>
<td>Clock Live Wallpaper</td>
<td>94.4</td>
<td>95.1</td>
<td>1.7</td>
<td>4.8</td>
<td>0.62</td>
</tr>
<tr>
<td>Multimedia</td>
<td>Ringtone generator</td>
<td>90.6</td>
<td>77.4</td>
<td>6.7</td>
<td>51.7</td>
<td>24.29</td>
</tr>
<tr>
<td>Multimedia</td>
<td>Signal Generator</td>
<td>98.9</td>
<td>97.3</td>
<td>78.6</td>
<td>96.4</td>
<td>8.95</td>
</tr>
<tr>
<td>Multimedia</td>
<td>Mytronomome</td>
<td>98.2</td>
<td>86.6</td>
<td>90.1</td>
<td>91.3</td>
<td>2.94</td>
</tr>
<tr>
<td>Multimedia</td>
<td>Instant Sound Effects</td>
<td>99.3</td>
<td>97.4</td>
<td>78.3</td>
<td>95.8</td>
<td>7.73</td>
</tr>
<tr>
<td>Security</td>
<td>Lock Pattern Generator</td>
<td>87.6</td>
<td>90.9</td>
<td>40.8</td>
<td>74.3</td>
<td>5.03</td>
</tr>
<tr>
<td>System</td>
<td>Battery Circle</td>
<td>86.9</td>
<td>89.0</td>
<td>51.3</td>
<td>58.2</td>
<td>1.69</td>
</tr>
<tr>
<td>System</td>
<td>Battery Dog</td>
<td>88.2</td>
<td>86.8</td>
<td>12.4</td>
<td>35.2</td>
<td>12.24</td>
</tr>
<tr>
<td>Time</td>
<td>World Clock</td>
<td>98.1</td>
<td>97.1</td>
<td>88.2</td>
<td>95.2</td>
<td>16.12</td>
</tr>
<tr>
<td>Time</td>
<td>Colour Clock</td>
<td>97.0</td>
<td>95.2</td>
<td>91.8</td>
<td>98.2</td>
<td>3.45</td>
</tr>
<tr>
<td>Writing</td>
<td>MemoPad</td>
<td>85.4</td>
<td>85.6</td>
<td>47.6</td>
<td>86.5</td>
<td>117.82</td>
</tr>
<tr>
<td>Writing</td>
<td>WikiAndPad</td>
<td>86.7</td>
<td>85.9</td>
<td>23.7</td>
<td>72.9</td>
<td>8.67</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>91.5</strong></td>
<td><strong>91.0</strong></td>
<td><strong>44.9</strong></td>
<td><strong>71.9</strong></td>
<td><strong>15.76</strong></td>
<td></td>
</tr>
</tbody>
</table>
positions on the UI. Also, the Monkey is not able to generate input relevant to the current app UI.

MobiGUITAR [5] is one of the automated UI-driven testing frameworks for Android apps, which is the most relevant to DroidDEV. In particular, MobiGUITAR implements a breadth-first and depth-first traversal algorithm for the app UI-model, and restarts the exercising procedure from the starting app state when it cannot find any new ones. In practice, a restart can be time-consuming for most of real-world Android apps. Also, MobiGUITAR requires major user efforts to manually configure multiple tool parameters [9].

A³E [2] aims to systematically discover new app UIs with high coverage via two techniques. Targeted exploration (a static approach) and Depth-first exploration (a dynamic approach), to improve activity and method coverage, respectively. Since A³E implements the standard depth-first exploration strategy for the app UI-model, it may overlook certain app UIs which may be located at deep levels in the app UI-model, and thus it would be suitable for the apps with simple finite state UI-models without loops where most of the app UIs are located at shallow depth in the app UI-model.

SwiftHand [3] aims to achieve code coverage quickly by learning and systematically exploring an abstraction of the app UI-model. It uses execution traces generated during the testing process to learn an approximate app UI-model, which is further used to choose inputs that lead to undiscovered app states. Also, it minimizes the number of app restarts by searching for paths to reach new app states using only UI inputs.

VII. CONCLUSION

We have proposed and implemented the DroidDEV tool, an automated app UI exerciser, to achieve high code coverage in automated functional UI testing of Android apps. The tool dynamically builds a finite UI flow graph for the app under test, injects UI-context-aware inputs into editable widgets, and generates UI events for non-editable widgets to automatically exercise app UIs. DroidDEV implements a best-first search algorithm to effectively and efficiently discover all the app UIs and exercise all their widgets. Our experimental results show that DroidDEV performs similarly to manual UI testing in terms of code coverage. More importantly, DroidDEV outperforms prevalent existing automated approaches such as MobiGUITAR and Monkey in terms of code coverage and exercising time.

REFERENCES


