Received: 1 April 2017 | Revised: 3 December 2017 | Accepted: 30 December 2017

DOI: 10.1002/spe.2564

EXTENDED CONFERENCE PAPER

WILEY

Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customated input generation

Yauhen Leanidavich Arnatovich[1] | Lipo Wang[1] | Ngoc Minh Ngo[2] | Charlie Soh[1]

[1]School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore

[2]Global Outreach and Extended Education, Arizona State University, Ho Chi Minh City, Vietnam

Correspondence
Lipo Wang, Department of Information Engineering, School of Electrical and Electronic Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798.
Email: elpwang@ntu.edu.sg

Summary

The increasingly prevalent use of mobile devices has raised the popularity of mobile applications. Therefore, automated testing of mobile applications has become an extremely important task. However, it is still a challenge to automatically generate tests with high coverage for mobile applications due to their specific nontrivial structure and the highly interactive nature of graphical user interfaces (GUIs). In this paper, we propose a novel automated GUI testing technique for mobile applications, namely, *Mobolic*. In this approach, tests with high coverage are automatically generated and executed by combining the online testing technique and customated input generation. Employing the online testing technique, *Mobolic* systematically explores the app GUI without falling in a loop. It generates relevant events "on the fly" that are followed by an immediate execution. In addition, involving the customated input generation, *Mobolic* automatically generates relevant user inputs such as user-predefined, concrete, or random ones. We implemented *Mobolic* and evaluated its performance on 10 real-world open-source Android applications. Our experimental results show the effectiveness and efficiency of *Mobolic* in terms of achieved code coverage and overall exercising time.

KEYWORDS

event-driven FSM, model-based testing, model checking, symbolic execution, systematic GUI exploration, textual input generation

1 | INTRODUCTION

In the last few years, there has been a distinct shift to mobile devices in numerous application areas such as email, social networking, entertainment, and e-commerce. This trend has prompted an explosive growth in the number and variety of mobile applications (often called "apps") being developed. As such, there is an increasing demand for automated testing techniques for mobile apps.

A central challenge in automated testing is the generation of tests with high coverage due to the highly interactive nature of the app graphical user interfaces (GUIs).[1,2] In addition, existing GUI testing techniques[3-6] for the automated testing of PC-based applications are no longer feasible to perform GUI testing of mobile apps due to their specific nontrivial GUI structure.[2,7] Consequently, in recent years, many automated GUI testing systems for mobile apps have been developed.[7-13]

However, in practice, they are still incapable of achieving high code coverage[13,14] because they fail to simultaneously address both of the following major problems.

- Mobile apps have a specific nontrivial GUI structure. As such, a large number of execution paths should be explored via the nontrivial app GUIs. Therefore, it has become impractical to automatically achieve high code coverage within a reasonable time.
- Mobile app GUIs are innately highly interactive because they commonly rely on sensible user inputs that are provided by humans. However, in practice, generating nontrivial user inputs in an automated manner is not an easy task.

To address the limitations of the existing GUI testing approaches, in this paper, we propose a novel automated GUI testing technique, namely, *Mobolic*. It aims to automatically generate tests with high coverage via the automated exercising of the mobile app GUIs with customated user inputs. *Mobolic* does this by integrating the online testing technique[15-17] and customated input generation into a single solution. It uses (1) the model checking technique and finite-state machine abstraction for the systematic exploration of the app GUI via the automated guiding of the exercising process and (2) customated input generation that involves user interface (UI) heuristics such as the textual values of the editable widget attributes to automatically generate relevant user-predefined or random inputs and involves symbolic execution to generate concrete user inputs.

Mobolic performs a functional automated GUI testing via an implementation of a black-box testing approach. It builds concrete app GUI models and uses them as input and generates relevant user inputs for automatic app exercising. As such, *Mobolic* innately belongs to the class of testing tools that perform automated GUI exploration via active learning. Therefore, since *Mobolic* performs automated app exercising via GUI exploration with active learning, it does not use the abstract app GUI models. Also, since *Mobolic* is a testing technique, it does not implement the capabilities of or intend to perform a software modeling[18,19] or software development[20,21] framework. As such, *Mobolic* does not facilitate the developers to examine various app GUI design alternatives, or evaluate many diverse configuration possibilities, or automatically generate source code, and others. Instead, *Mobolic* aims to automate the GUI testing process by minimizing or eliminating manual efforts.

Mobolic automatically guides the testing procedure by dynamically building a finite-state model of the app GUI without loops.[22] This is done by extending Google's UI Automator testing framework.* The UI Automator testing framework is designed for writing black-box–style functional automated UI tests, where the test code does not rely on the internal implementation details of the target app. Other than the UI Automator, there are other automated UI testing frameworks such as MonkeyRunner,† Troyd,‡ Robotium,§ Appium,¶ and Android GUITAR.# Unlike these, the UI Automator provides a unique feature to dynamically dump a view hierarchy of the foreground app screen displaying on the Android device into XML files. From XML dumps, *Mobolic* builds a finite-state model of the app GUI without loops and, thus, guides the testing procedure.

Mobolic explores the app GUI using our novel *f-GFG* model (see Sections 2.2 and 2.3). It automatically builds the model and systematically traverses it by implementing an informed search algorithm A*, which is the best-known form of the best-first search (BFS) algorithm.[23] The A* search algorithm combines BFS for efficiency with the uniform cost search for optimality and completeness. The key idea behind the A* search algorithm is to find the shortest path leading to the target app UI (see Section 2.5.3). For more details, refer to Section 6.1, where we discuss related systematic GUI exploration strategies that are based on different search algorithms.

Mobolic exercises app by automatically and systematically interacting with the app UI widgets. However, due to the highly interactive nature of the app GUI, it commonly requires specific user inputs that are practically impossible to generate randomly. Therefore, to generate such specific user inputs, *Mobolic* implements our novel customated input generation mechanism (see Section 2.3). It allows *Mobolic* to generate realistic user inputs that are predefined by the user for certain input types as described in Section 2.5.5. In addition, the customated input generation mechanism allows *Mobolic* to generate concrete user inputs. For that reason, *Mobolic* uses the symbolic execution technique. However, there are situations when the user-specific or concrete user inputs cannot be generated, and thus, to fully automate the exercising process,

*http://developer.android.com/tools/help/uiautomator/index.html
†http://developer.android.com/tools/help/monkeyrunner_concepts.html
‡https://github.com/plum-umd/troyd
§http://code.google.com/p/robotium
¶http://appium.io
http://sourceforge.net/projects/guitar

Mobolic supports random-input generation. As a result, *Mobolic* explores the app GUI without human interaction. For more details, refer to Section 6.2, where we discuss concrete-input generation via symbolic execution.

In this paper, we made the following contributions.

- We propose a novel automated GUI testing technique, namely, *Mobolic*, which combines the online testing technique and customated input generation to generate tests with high coverage.
- We implement *Mobolic* for Android apps, taking into account their specific nontrivial structure and the highly interactive nature of GUIs.
- We evaluate the performance of *Mobolic* on 10 real-world open-source Android apps and compare it with prevalent approaches such as manual exercising (Human), Sapienz, ACTEve, and MobiGUITAR.

In Section 2, we state the major problems and describe the proposed approach by giving definitions, an overview, and detailed explanations of *Mobolic* components. In Section 3, we describe the design and implementation of *Mobolic* by sharing our practical experience obtained during *Mobolic* development. In Section 4, we give a demo of the *Mobolic* workflow by using a concrete example of the app GUI. In Section 5, we evaluate the performance of *Mobolic* by comparing it with prevalent approaches such as Human, Sapienz, ACTEve, and MobiGUITAR. In Section 6, we discuss the *Mobolic* approach. We discuss its novel and improved features by describing the principal differences between *Mobolic* and traditional model-based testing and between *Mobolic* and automated input generation via symbolic execution. In Section 7, we offer an overview of related work by describing state-of-the-art automated GUI testing approaches such as Monkey, Dynodroid, MobiGUITAR, A^{3}E, *SwiftHand*, Sapienz, and ACTEve. In Section 8, we conclude our work and set future research directions.

2 | PROPOSED APPROACH: MOBOLIC

In this section, we describe the related problems and give the definitions, an overview, and details of *Mobolic*. In Section 2.1, we state 2 main problems that *Mobolic* addresses. In Section 2.2, for *Mobolic* explanations, we give several conceptual definitions. In Section 2.3, we describe the novel aspects of *Mobolic*. In Section 2.4, we introduce *Mobolic* by briefly describing its components and walking through the overall exercising procedure. In Section 2.5, we describe in detail each component of *Mobolic* by explaining their working mechanisms.

2.1 | Problem statement

There are 2 fundamental problems, in the literature, of automated GUI testing for Android apps, which have been well highlighted by researchers in the field.[7,9-12,24-27] The first problem, **Problem#1**, is related to the GUI exploration strategy through model-based testing, and the second problem, **Problem#2**, is related to concrete-input generation through the symbolic execution technique.

Problem#1. A majority of the mobile apps have a nontrivial GUI structure, ie, their UI models are not finite state and/or contain loops. Therefore, it is a challenge to perform automated GUI testing on these apps. For example, the existing tools for GUI testing of mobile apps build the GUI model and use it "as is" for systematic exploration. However, in fact, the GUI models of mobile apps may innately have an infinite number of UIs or may have infinite loops. As such, the GUI testing procedure may infinitely exercise app unless the termination condition is explicitly specified by the user.[15] Hence, to address this problem, the existing tools set the termination condition for the testing procedure such as limited execution time, or manually predefined number of events to be injected, or the depth of exploration in the UI model.[14,15] Therefore, with such specified termination conditions, the existing tools may not discover certain app UIs, especially if they are located at deep levels in the UI model.

Problem#2. Mobile app GUIs are commonly highly interactive so that the apps require sensible input from the users. As such, to automatically exercise the app, it is crucial that nontrivial user inputs must be automatically generated to feed into the app. To the best of our knowledge, this remains a challenging task. For example, many existing tools[14] for the GUI testing of mobile apps either generate inputs randomly and/or provide them manually during testing. However, random inputs are trivial, whereas manual efforts are time consuming. Therefore, to automatically generate relevant user inputs, existing tools[8,28,29] have applied symbolic execution. These tools tend to symbolically execute all program paths existing in the app. However, symbolic execution suffers from limitations such as *path explosion*, *path divergence*, and *constraint complexity*.[30] Therefore, to execute all program paths symbolically is impractical for all but trivial apps. Thus, the search of relevant inputs needs to be either *depth*-bounded or *time*-capped.[31,32]

2.2 | Definitions

For the explanations of *Mobolic*, in this section, we give 5 definitions that are used throughout this paper.

Definition 1. ***User interface (UI)*** is a view hierarchy of the objects on the app screen, which defines its layout. The objects may be input controls (editable) or other widgets (noneditable).

This definition states that each UI usually has one or more UI widgets, each could be editable (eg, "*EditText*") or noneditable (eg, "*Button*"). A noneditable UI widget allows a transition to the same or another UI. Within the context of this paper, we consider that 2 UIs are equivalent if the number of UI widgets and their values of the attribute "*class*" in the corresponding view hierarchies match. The UI widget attributes "*text*" (only for editable widgets), "*NAF*," "*bounds*," "*focusable*," "*enabled*," "*checkable*," "*scrollable*," "*long-clickable*," "*selected*," "*focused*," "*clickable*," and "*checked*" are excluded from the comparison since these attributes do not imply the equivalence of 2 UIs.

Definition 2. ***GUI flow graph (GFG)*** is defined as a triple $g = (U,W,T)$, where U is the set of nonequivalent UIs, W is the set of widgets on a UI, and T is the set of transitions, where each transition t_i is a directed edge from a widget w_{ij} to UI u_q, denoted by $t_i = w_{ij} \rightarrow u_q$, where $t_i \in T$, $w_{ij} \in W$, $u_q \in U$ and w_{ij} is on u_q, where "i" is the index of the UI in *GFG*, and "j" is the index of the widget on the UI, where $i > 0, j > 0, 0 < q \leq i + 1$.

An example of *GFG* is shown in Figure 1. In Figure 1, u_i represents a UI, e_{ij} represents an editable widget, and w_{ij} represents a noneditable widget, where "i" is the sequential index of the UI, and "j" is the sequential index of the widget on the UI. On each UI, w_{ij} allows a transition to the same or another UI, as indicated by the arrows.

In Figure 1, *GFG* is finite state; however, it has 2 loops. For example, the sequence of transitions $t_1 = w_{12} \rightarrow u_2$, $t_2 = w_{21} \rightarrow u_3$, and $t_3 = w_{31} \rightarrow u_1$ forms one loop $u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_1$, and another sequence of transitions $t_1 = w_{12} \rightarrow u_2$, $t_2 = w_{21} \rightarrow u_3$, and $t_3 = w_{32} \rightarrow u_3$ forms another loop $u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_3$. Therefore, we need to build a *GFG* that is always finite state and does not have any loops.

Definition 3. ***Finite-state GFG (f-GFG)*** is a GFG that is finite state and does not have any sequence of transitions $t_i, t_{i+1}, \ldots, t_n$, which forms a *loop* from $u_i, u_{i+1}, \ldots, u_m$, where $t_n = w_{nj} \rightarrow u_m$, $t_n \in T$, $u_m \in U$, and $i > 0, j > 0, n > 0, m > 0$, $i \leq n, i \leq m, m \leq n$.

Definition 4. ***User input–dependent statement (UID statement)*** is a statement that is control and/or data dependent[33,34] on one or more input values provided by the user through the app GUI.

The example of code in Figure 2 gives an illustration of UID statements. From Figure 2, we can see that UID statements are in lines 2 and 7, whereas the non–UID statement is in line 14.

Definition 5. ***Reduced control dependence graph (r-CDG)*** is a subgraph of a conventional CDG, which consists of branches, where at least one decision node corresponds to the UID conditional statement.

In Figure 3, we give an example of r-CDG, which is obtained from the app code shown in Figure 2. In Figure 3, nodes ①, ②, ④, ⑤, and ⑥ correspond to code lines 2, 3-4, 7, 8-9, and 11-12, respectively, as shown in Figure 2. In Figure 3, decision nodes ① and ④ (ie, "if" statements) are shaded, and *T* and *F* abbreviations denote "True" and "False" branches, respectively.

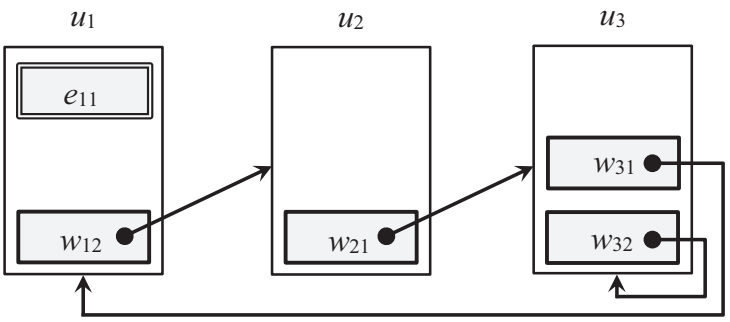

FIGURE 1 Example of graphical user interface flow graph for app

```
1:  void foo(int x) { /* "x" contains user value */
2:    if (x > 5) {
3:      /* to discover u2, u3 */
4:      /* to execute code#1 */
5:    }
6:    else {
7:      if (x == 5) {
8:       /* to execute code#2 */
9:       return;
10:     }
11:     /* to discover u1 */
12:     /* to execute code#3 */
13:   }
14:   if (getSystemTime() == '00:00:00') {
15:     /* to execute code#4 */
16:   }
17: }
```

FIGURE 2 Example of app code with user input-dependent statements

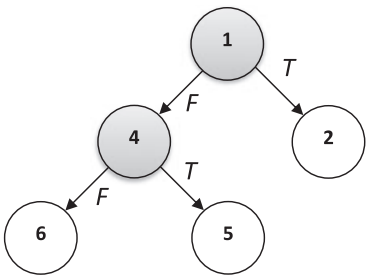

FIGURE 3 Example of reduced control dependence graph for app code

2.3 | Novel aspects of Mobolic

In this section, we show the novel aspects of *Mobolic*, which are implemented via various improvements and/or new features that existing tools either do not implement in the same fashion and/or simply do not have. For clarity, we summarize the novel aspects of *Mobolic* as follows.

1. In *Mobolic*, we introduce a new feature such as the ***f-GFG*** model of the app GUI. The *f-GFG* model is a finite-state GUI model that does not have loops at any point in time. The *f-GFG* model enables *Mobolic* to have the following improvements.
   (a) It enables *Mobolic* to notably reduce exercising time. Instead of using a standard implementation of *depth*-first or *breadth*-first search or random GUI exploration algorithms, *Mobolic* implements an ad hoc GUI exploration strategy that is based on the existing A* technique (see Section 3.1). It allows *Mobolic*, during the exercising process, to quickly find the shortest path to the lastly discovered app UI from which the exercising process continues. As such, the overall exercising time is reduced, which makes *Mobolic* more efficient in comparison with the other automated approaches discussed in the paper.
   (b) It enables *Mobolic* to notably increase efficacy of the automated GUI exploration. Without falling into an infinite loop, *Mobolic* discovers as many UIs existing in the app GUI model as possible. As such, *Mobolic* demonstrates significantly higher code coverage results in comparison with the other automated approaches discussed in the paper.

2. In *Mobolic*, we introduce a new feature such as the ***customated input generation*** mechanism. Customated input generation enables *Mobolic* to have the following improvements.

   (a) It enables *Mobolic* to generate concrete user inputs while mitigating the well-known problems of the conventional symbolic execution. The idea is to symbolically execute only UID statements since the nature of the mobile app GUIs is highly interactive and a majority of the textual inputs are, in general, expected from the user. Hence, it allows *Mobolic* to avoid extra overhead of symbolically executing all existing in the app conditional statements, and thus, *Mobolic* finishes the exercising process within a reasonable amount of time.
   (b) It enables *Mobolic* to generate UI-context-aware user inputs. *Mobolic* uses a manually crafted dictionary of basic text terms, which are, in turn, built on the textual terms extracted from multiple reputable apps. The idea of the UI-context-aware inputs is to automatically, during the exercising process, obtain relevant input values that lie in a set of possible values with a specific input pattern, eg, email address or server IP address. Therefore, it allows *Mobolic* to automatically resolve such situations where the user inputs with a specific pattern are required, and thus, *Mobolic* avoids human interaction in the testing process.

2.4 | Overview of Mobolic

In this section, we introduce the components of *Mobolic*, give an overview of the *Mobolic* workflow, and show an overall testing procedure of *Mobolic*. The detailed explanations on *Mobolic* components are given in Section 2.5.

Mobolic consists of 5 main components.

- **Static Analyzer (stAnalyzer)**. This component builds and analyzes the control dependence graph (CDG) and the data dependence graph (DDG) for the original Android app. Using the CDG and the DDG, it builds the r-CDG for the mobile app. As an output, it returns the built r-CDG (see Section 2.5.1).
- **Application Instrumenter (appInster)**. Guided by the r-CDG, this component instruments UID conditional statements in the app so that the corresponding UID symbolic constraints can be collected during app execution. As an output, it returns an instrumented app (see Section 2.5.2).
- **Systematic Automated Test Driver (SAT)**. This component automatically and systematically explores the app GUI by interacting with UI widgets that are displayed on the mobile device screen. As an output, it returns a flag, which indicates whether the exercising procedure falls into a loop (see Section 2.5.3).
- **Input Scheduler (iScheduler)**. This component forms UID symbolic path constraints (PCs) leading to yet unexplored execution paths so that new app UIs can be discovered and exercised. As an output, it returns a symbolic PC, which is formed for a particular editable widget (see Section 2.5.4).

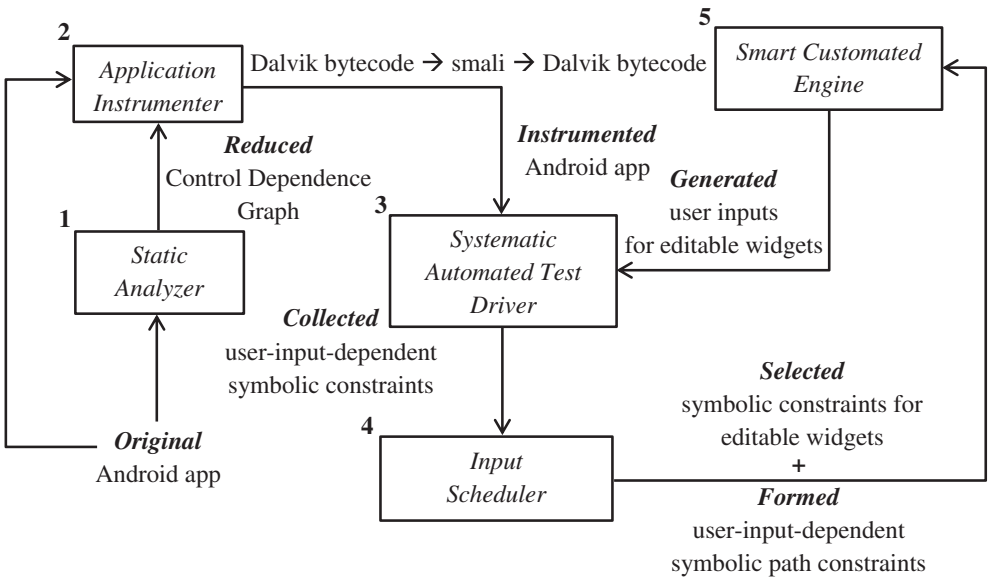

FIGURE 4 Overview of the *Mobolic* workflow

- **Smart Customated Engine (smartCE)**. This component generates relevant specific user inputs by finding user-predefined inputs, or solving UID symbolic PCs, or generating random inputs. As a result, app UIs that are constrained by such specific user inputs can be explored. As an output, it returns relevant user inputs for the editable widgets (see Section 2.5.5).

Here, we overview the *Mobolic* workflow, which is shown in Figure 4. *Mobolic* first runs *Static Analyzer* (*stAnalyzer*) to perform a static analysis of the original Android app and build the DDG, CDG, and r-CDG. Guided by the r-CDG, *Application Instrumenter* (*appInster*) instruments UID conditional statements. As a result, *appInster* rebuilds the Android app with instrumented UID conditional statements. *Systematic Automated Test Driver* (*SAT*) exercises the instrumented app by automatically and systematically exploring the app GUI. Also, during the exercising process, through the app instrumentation, *SAT* collects UID symbolic constraints over the instrumented UID conditional statements along the execution path. From the collected UID symbolic constraints, *Input Scheduler* (*iScheduler*) forms relevant UID symbolic PCs for the selected editable widget, to explore yet uncovered execution paths. *Smart Customated Engine* (*smartCE*) generates relevant user inputs by solving the UID symbolic PCs with concrete values, generating user-predefined or random ones.

Next, we show the overall workflow of *Mobolic* in Algorithm 1. Generally, *Mobolic* can be logically divided into 3 stages: (1) preparation (lines 2-6), (2) input generation (lines 9-20), and (3) exercising of the app under test (lines 21-34).

Stage#1. *Mobolic* performs static analysis and instrumentation of the app, initializes variables, and runs the app. In line 2, *stAnalyzer* performs static analysis of the app and builds CDG and DDG with the aid of the Amandroid[35] tool. As an output, *stAnalyzer* returns (r-CDG). Guided by the r-CDG, in line 3, *appInster* creates an instrumented app (*instApp*) by replacing the original UID conditional statements in the *app* with instrumented ones. In lines 4 and 5, *Mobolic* initializes the set of fully exercised UIs (*feui*) and the *f-GFG* as empty. The *f-GFG* is used to guide the exercising procedure and to ensure that *Mobolic* does not fall in an infinite loop, and *feui* is used for memorizing fully exercised UIs. The *feui* and *f-GFG* are used together for the termination condition. *Mobolic* terminates the exercising procedure once all the discovered UIs are fully exercised. Next, *Mobolic* runs the instrumented app *instApp* (line 6).

It is important to note that *Mobolic* determines that an app UI is *fully exercised* only if (1) all its relevant noneditable widgets (if any) are fully exercised and (2) all the decision nodes for each editable widget (if any) on the app UI are fully covered. In turn, *Mobolic* determines that a noneditable widget is *fully exercised* only if all its relevant UI events are executed. In addition, *Mobolic* determines that a decision node for the editable widget is *fully covered* only if its corresponding UID statement with all its child subconditional statements are executed.

Stage#2. *Mobolic* performs input generation. After analyzing, instrumenting, and running the instrumented app *instApp* in Stage#1, *Mobolic* first obtains the current UI (*ui*) displaying on the Android device (line 8). Next, in line 10, *iScheduler* forms a relevant symbolic PC for the selected editable widget (*ew*) by either appending the newly collected UID symbolic constraints to the existing symbolic PC or negating the last decision node of the existing symbolic PC. In line 11, *Mobolic* obtains a user input by executing *smartCE*, a customated input generation mechanism. First, *smartCE* searches for a user-predefined input matching textual terms of the attributes "*text*," "*resource-id*," and "*content-desc*" for the selected editable widget *ew*. If not found (eg, input type is not recognized or does not exist), it searches for a default value that is assigned by the app itself. If the default value does not exist, it tries to solve the formed symbolic PC. If the formed symbolic PC (*pc*) cannot be solved (eg, infeasible path) or *pc* is not formed yet (eg, when *ew* is exercised for the first time), *smartCE* generates a random user input.

Mobolic injects the generated user input *in* into the selected editable widget *ew* (eg, sets text for "*EditText*" widget) (line 12). In line 13, *Mobolic* checks if the symbolic PC *pc* has been formed. If so, *Mobolic* extracts the last decision node (*ldn*) from the formed symbolic PC *pc* (line 14). If *ldn* is fully covered (line 15), *Mobolic* removes it from *pc* (line 16). Next, for the selected *ew*, *Mobolic* updates the symbolic UID constraints by replacing the previously stored ones with the constraints from the newly formed symbolic PC *pc* (line 19).

Stage#3. *Mobolic* exercises the app until all the discovered UIs in *f-GFG* are fully exercised. After input injection in Stage#2, *Mobolic* first obtains a flag (*infloop*), which indicates a loop occurrence in the exercising procedure, by executing *SAT* (line 21). For that purpose, *SAT* retrieves a noneditable unexercised relevant widget from the current UI *ui*. Afterward, depending on the type of the widget, it generates a relevant UI event to exercise the widget rendering a subsequent UI. Also, during exercising, *SAT* stores symbolic UID constraints for each editable widget, which are encountered on the execution path. Next, *SAT* builds *f-GFG* by adding the rendered subsequent UI with its widgets into *f-GFG* only if the rendered subsequent UI does not exist in *f-GFG* at this point in time. Note that exercising a noneditable widget could render a new subsequent UI. If so, *SAT* memorizes such a link to store the transition from the exercised widget to its rendered subsequent UI. As such, *SAT* ensures that *f-GFG* does not have any loop, and *Mobolic* will not exercise the app infinitely.

Algorithm 1: Mobolic

```
Input: app – app under test
1  procedure Mobolic(app)
2      r-CDG = stAnalyzer(app);
3      instApp = appInster(app, r-CDG);
4      feui = < empty >;
5      f-GFG = < empty >;
6      Run(instApp);
7      while (TRUE) do
8          ui = GetCurrentUI();
9          foreach (editable widget ew ∈ ui) do
10             pc = iScheduler(ew);
11             in = smartCE(pc, ew);
12             InjectInput(in, ew);
13             if (pc.isFormed()) then
14                 ldn = GetLastDecisionNode(pc);
15                 if (ldn.isFullyCovered()) then
16                     pc.Remove(ldn);
17                 end
18             end
19             UpdateStoredUIDConstraints(pc, ew);
20         end
21         infloop = SAT(f-GFG, ui);
22         lui = GetLastlyDiscoveredUI(f-GFG);
23         if (lui.isFullyExercised()) then
24             feui.Add(lui);
25             f-GFG.Remove(lui);
26             if (lui.isCurrentUI()) then
27                 PressBack();
28             end
29         end
30         if (infloop == TRUE) then
31             ui = GetCurrentUI();
32             lp = FindPathToLastlyDiscoveredUI(f-GFG, ui);
33             ReplayPathToLastlyDiscoveredUI(lp);
34         end
35         if (f-GFG ∈ feui) then
36             break while;
37         end
38     end
39 end
```

Mobolic obtains the lastly discovered UI (*lui*) from *f-GFG* (line 22). The lastly discovered UI is the UI that is lastly added into *f-GFG*. Next, *Mobolic* checks if *lui* is fully exercised (line 23). If so, *Mobolic* adds *lui* into the set of fully exercised UIs (*feui*) (line 24), removes it from *f-GFG*, and changes the lastly discovered UI in *f-GFG* (line 25). In addition, if *lui* is

a current UI (line 26), *Mobolic* generates a system key event "*Back*" (line 27), because *lui* has been fully exercised and removed from *f-GFG*. This event may return to any previously discovered app UI existing in *f-GFG*. Note that, depending on the app logic, system key event "*Back*" may exit the app. Such case is handled by *Mobolic* by restarting the app and continuing the exercising procedure.

If *SAT* indicates that a loop has occurred in the exercising procedure, ie, *infloop* is TRUE (line 30), *Mobolic* obtains the current UI *ui* (line 31), finds a path (*lp*) from *ui* to the lastly discovered UI in *f-GFG* (line 32), and replays *lp* to reach the lastly discovered UI (line 33). As a result, at the next iteration, the lastly discovered UI will be appear as the current UI *ui* displaying on the device screen (line 8), and the exercising procedure continues. The *FindPathToLastlyDiscoveredUI* function (line 32) performs A* search by using the stored links in *f-GFG* to find the shortest path leading to the lastly discovered UI in *f-GFG*. The *ReplayPathToLastlyDiscoveredUI* function (line 33) interacts with all noneditable widgets and generates user inputs for all editable widgets along the path *lp* to reach the lastly discovered UI in *f-GFG*. Therefore, if there is any loop occurrence in the exercising procedure, *Mobolic* can easily track back the lastly discovered UI and continue exercising yet unexercised widgets. *Mobolic* automatically terminates the exercising process once all discovered app UIs are fully exercised (ie, the set of fully exercised UIs *feui* contains all UIs from *f-GFG*) (lines 35-37).

2.5 | Components of Mobolic

In this section, we describe, in detail, the functionality of the *Mobolic* components and walk through their corresponding algorithms.

2.5.1 | Static Analyzer (*stAnalyzer*)

To identify UID statements, *stAnalyzer* performs a user input dependence analysis on the CDG and DDG[33,34] of the app. In particular, *stAnalyzer* builds the graphs with the aid of a data flow analysis framework for Android apps, namely, Amandroid.[35] Note that, however, Amandroid does not innately solve the problem of relating Java code with the editable widgets on the UI. To identify the UID statements, we implement our solution that leverages intermediate representations Jawa[‖] and smali[**] from Amandroid and Apktool,[††] respectively.

The CDG is used to track intra- and inter-procedural conditional statements. From the CDG, *stAnalyzer* builds r-CDG to eliminate all the conditional statements that are irrelevant toward the reachability of the target UID conditional statement. The DDG is used to track the changes made to the UID statements that are data dependent on the user inputs. Therefore, all the changes can be reflected into the UID conditional statements and solved for symbolic user variables so that the corresponding concrete user inputs can be found.

In Algorithm 2, we show the main steps of how *stAnalyzer* performs static analysis of the app. First, *stAnalyzer* initializes the r-CDG as empty (line 2) and builds *CDG* and *DDG* (lines 3-4). Next, guided by *DDG*, for each branch (*b*) in *CDG* (line 5) and for each decision node (*d*) in *b* (line 6), *stAnalyzer* checks if *d* has any data dependency on the user input (line 7). If so, *stAnalyzer* adds the selected branch *b* into r-CDG (line 8). The *stAnalyzer* terminates once all branches in *CDG* are analyzed, and it returns r-CDG (line 13).

2.5.2 | Application Instrumenter (*appInster*)

Guided by r-CDG, *appInster* performs the instrumentation of UID conditional statements. Therefore, the instrumented UID conditional statements can be collected. Note that in UID conditional statements, user input–*in*dependent symbolic values will be replaced with the actual ones that are assigned by the app itself during runtime. The instrumented UID conditional statements may include "*if-else*," "*switch-case*," "*for*," and "*while-loop*."

There are several instrumentation techniques.[36] The *appInster* implements the most common one, ie, bytecode instrumentation. Technically, *appInster* uses smali intermediate language that represents *Dalvik* bytecode in readable form. The *appInster* instruments UID conditional statements by inserting additional probes (code) into smali that is obtained with the aid of Apktool, a tool for reverse engineering of Android apps. After instrumentation, *appInster* assembles the Android app from the instrumented smali code so that the app can be normally installed and executed.

[‖] http://pag.arguslab.org/jawa-language
[**] https://github.com/JesusFreke/smali
[††] http://ibotpeaches.github.io/Apktool/

Algorithm 2: stAnalyzer

```
Input: app – app under test
Output: r-CDG – reduced control dependence graph
1  procedure stAnalyzer(app)
2      r-CDG = < empty >;
3      CDG = BuildCDG(app);
4      DDG = BuildDDG(app);
5      foreach (branch b ∈ CDG) do
6          foreach (decision node d ∈ b) do
7              if (d.isDataDependentOnUserInput(DDG)) then
8                  r-CDG.Add(b);
9                  break;
10             end
11         end
12     end
13     return r-CDG;
14 end
```

In Algorithm 3, we show the main steps of how *appInster* instruments the app. Guided by the r-CDG, for each branch (b) in r-CDG (line 2) and for each decision node (d) in b (line 3), *appInster* instruments d to obtain an instrumented decision node (id) (line 4). Afterward, it replaces d with id in original (app) and generates an instrumented app ($instApp$) (line 5). The *appInster* terminates once all branches in r-CDG are processed and returns the instrumented app $instApp$ (line 8).

Algorithm 3: appInster

```
Input: app – original app, r-CDG – reduced control dependent graph
Output: instApp – instrumented app
1  procedure appInster(app, r-CDG)
2      foreach (branch b ∈ r-CDG) do
3          foreach (decision node d ∈ b) do
4              id = InstumentDecisionNode(d);
5              instApp = ReplaceDecisionNode(d, id, app);
6          end
7      end
8      return instApp;
9  end
```

2.5.3 | Systematic Automated Test Driver (*SAT*)

SAT is primarily designed for performing black-box–style functional automated UI testing, where the test code does not rely on the internal implementation details of the target app. *SAT* implies an event-driven automated UI test driver. It automatically and systematically explores the app GUI by interacting with noneditable UI widgets that are displayed on the Android device screen. In particular, *SAT* exercises all relevant UI widgets one by one, in accordance with a sequence in which widgets are innately located in the UI hierarchy view, from up to down and from left to right. To qualify for the systematic GUI exploration, *SAT* must ensure that the exercising process is not interrupted. In fact, systematic GUI exploration is commonly interrupted by loops existing in the GUI flow. Thus, to address this issue and to ensure that *SAT* performs systematic GUI exploration, *Mobolic* implements the A* search algorithm. For example, if a loop occurs during

TABLE 1 Types of events supported by *Mobolic* Systematic Automated Test Driver

Event Origin: Type	Actions	Hardware Keys
UI: Touch	Click; LongClick	–
UI: Motion	Swipe; Scroll; Pinch	–
UI: Trackball	Roll; Press	–
UI: Keypress	Input Injection	–
System: "Major" navigation	–	"Home"; "Back"; "Menu"

exercising, in order to return to the app UI on which the exercising process has been interrupted, *Mobolic* performs A* search in *f-GFG* to find the shortest path from the current app UI, displaying on the Android device screen, to the lastly discovered app UI. In particular, once the loop has occurred, *SAT* indicates an interruption in the exercising process, ie, it returns TRUE to the exercising procedure, by which it signals *Mobolic* to perform A* search. As a result, guided by the A* search, *Mobolic* returns the exercising procedure to the lastly discovered app UI so that *SAT* is be able to continue with the GUI exploration from that lastly discovered app UI.

During exercising, *SAT* automatically (1) builds the *f-GFG*, which is used to continuously guide the exercising procedure without falling in a loop, and (2) collects UID symbolic constraints along the execution path through the app instrumentation. Note that *SAT* collects UID symbolic constraints over UID conditional statements once their corresponding UID conditional statements are encountered (ie, their code is executed). For the comprehensive exercising of app GUI, *Mobolic* supports common UI user actions that are listed in Table 1.

Algorithm 4: SAT

Input: *f-GFG* – finite state GUI flow graph, *ui* – current UI
Output: *infloop* – flag of loop occurrence

```
1  procedure SAT(f-GFG, ui)
2      infloop = TRUE;
3      if (ui ∉ f-GFG) then
4          f-GFG.Add(ui);
5      end
6      nw = GetUnExercisedRelevantWidget(ui);
7      ExerciseNonEditableWidget(nw);
8      foreach (editable widget ew ∈ ui) do
9          StoreEncounteredUIDConstraints(ew);
10     end
11     ui = GetCurrentUI();
12     if (ui ∉ f-GFG) then
13         f-GFG.Add(ui);
14         f-GFG.AddLink(nw, ui);
15         infloop = FALSE;
16     end
17     return infloop;
18 end
```

In Algorithm 4, we show the main steps of how *SAT* guides an exercising procedure by building *f-GFG* and collecting UID symbolic constraints. *SAT* first sets the flag of loop occurrence (*infloop*) to be TRUE (line 2) and adds the current UI (*ui*) with its widgets into *f-GFG* (line 4) if *ui* is not yet in *f-GFG* (line 3). In line 6, *SAT* obtains a relevant unexercised

noneditable widget (*nw*). Next, *SAT* exercises the relevant widget by generating a relevant UI event for the widget *nw* (line 7). Note that to generate the relevant UI event, *SAT* analyzes the attributes of the widget *nw*. Firing the UI event upon *nw* executes certain code in the app. As such, for each editable widget *ew* on the current UI *ui*, *SAT* stores all UID symbolic constraints encountered on the execution path (lines 8-10). By firing a UI event upon the widget, the current UI could be changed. As such, the current UI *ui* must be updated (line 11). If the current UI *ui* is not in *f-GFG* (line 12) (ie, *ui* is a newly discovered UI), *SAT* adds *ui* with its widgets into *f-GFG* (line 13) and adds a link (*nw*,*ui*) into *f-GFG* (line 14) to memorize the transition from *nw* to *ui*. In addition, *SAT* sets the flag *infloop* to FALSE (line 15). In line 17, *SAT* returns the *infloop* flag to *Mobolic*. If the returned flag *infloop* is FALSE, it signals *Mobolic* that there is no loop occurrence in the exercising procedure, and thus, *SAT* can continue with the newly discovered UI. Otherwise, if the returned flag *infloop* is TRUE, it signals *Mobolic* that there is a loop occurrence in the exercising procedure, and thus, *SAT* requires *Mobolic* to find the lastly discovered UI to continue the exercising procedure.

2.5.4 | Input Scheduler (*iScheduler*)

This component forms UID symbolic PCs based on the UID symbolic constraints collected by *SAT*. The *iScheduler* ensures that the relevant UID symbolic PCs are formed for every editable widget to cover as many execution paths as possible. *Mobolic* is an iterative process, and for every iteration, the newly generated user inputs may lead to unexplored program paths with different app functionalities. As such, to explore as many execution paths as possible, *iScheduler* derives a UID symbolic PC in each iteration. In particular, *iScheduler* forms a UID symbolic PC by tweaking collected UID symbolic constraints as follows.

1. For an "*if*" statement, its corresponding UID symbolic constraint is constructed. The constructed UID symbolic constraint is then appended to the earlier formed UID symbolic PC of the nearest ancestor UID conditional statement to form the symbolic PC of the "*if*" statement. Note that if the ancestor UID conditional statement does not exist for a particular "*if*" statement, the UID symbolic PC will only consist of the UID symbolic constraint corresponding to its own "*if*" statement.
2. For an "*else*" statement, its UID symbolic constraint is constructed from the corresponding "*if*" statement by applying the logical negation operator "NOT." The constructed UID symbolic constraint is then appended to the earlier formed UID symbolic PC of the nearest ancestor UID conditional statement to form the symbolic PC of the "*else*" statement.

In practice, for all but trivial apps, the user inputs may be required on any UI of the app; therefore, the user inputs commonly depend on each other. Considering such dependencies, *iScheduler* uses the following rules to eliminate missing any yet undiscovered app UIs.

1. For each editable UI widget, at a time, *iScheduler* forms only 1 relevant symbolic PC since each editable UI widget may have several distinct symbolic PCs depending on the app logic.
2. For each editable UI widget, at a time, *iScheduler* forms the symbolic PC whose solution (ie, generated user input) may contribute to discovering yet unexplored app UIs.

In Algorithm 5, we show the main steps of how *iScheduler* forms a relevant symbolic PC for a selected editable widget. First, *iScheduler* sets symbolic PC (*pc*) to empty (line 2) and obtains all UID symbolic constraints (*sc*) stored by *SAT* for the selected editable widget (*ew*) (line 3). Next, *iScheduler* checks if the obtained UID symbolic constraints *sc* are not empty (line 4). If so, from *sc*, *iScheduler* forms a symbolic PC *pc* (line 5). If *sc* does not have any new symbolic UID constraints (ie, the constraints that are not included into the returned *pc* at the previous iterations) (line 6), *iScheduler* negates the last decision node in the formed symbolic PC *pc* (line 7). As a result, *iScheduler* forms and returns a relevant symbolic PC whose solution (ie, generated user input) may lead to discovering yet unexplored app UIs (line 10).

2.5.5 | Smart Customated Engine (*smartCE*)

The *smartCE* is a customated engine that is used to automatically generate specific user inputs. The *smartCE* implements our novel customated input generation mechanism that handles user-predefined (via exploiting a particular textual attributes of the editable UI widgets), concrete (via solving symbolic PCs that are collected through the app instrumentation), and random (numerical or textual) inputs.

Algorithm 5: iScheduler

```
Input: ew – editable widget
Output: pc – symbolic path constraint
1  procedure iScheduler(ew)
2      pc = < empty >;
3      sc = GetStoredUIDConstraints(ew);
4      if (sc.isNotEmpty()) then
5          pc = FormSymbolicPathConstrain(sc);
6          if (sc.hasNotNewSymbolicConstraints()) then
7              NegateLastDecisionNode(pc);
8          end
9      end
10     return pc;
11 end
```

For the user-predefined input generation, we build *Dictionary of Basic Text Terms*, namely, DBT2, which is a part of our customated input generation mechanism. DBT2 consists of the most commonly appeared input types such as "*email*," "*password*," "*login*," "*server*," "*port*," "*URL*," and "*phone*." To build DBT2, we manually inspected selected apps from *Google Play*, which require complex inputs, eg, *Skype*, *Google Chrome*, *WhatsApp*, *Facebook*, *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 we selected apps from the trusted developers, we believe that text terms of the selected attributes adequately describe the actual meaning of the required input types. Thus, we exploit the text terms of the UI attributes to build our own dictionary. We emphasize that DBT2 consists of the input types and their corresponding default valid input values. For example, for an editable UI widget that requires the *email* input type, DBT2 defines a default valid input value "*email@example.com*." However, optionally, for the input types in DBT2, the realistic textual input values can be provided by the user in the "*config.properties*" file so that they will be automatically injected during app exercising instead of the default ones.

For concrete-input generation, the *smartCE* takes the UID symbolic PC formed by *iScheduler* and solves it to obtain a concrete user input. Note that since symbolic execution is naturally time consuming, the found user input values are stored for every editable widget and could be further reused in the following exercising to shorten the exercising time by not solving the same UID symbolic PCs multiple times. Solving the UID symbolic PC, *smartCE* finds concrete user inputs that satisfy a particular symbolic PC corresponding to the execution path in the app code. Therefore, concrete user inputs allow *Mobolic* to discover app UIs that are constrained by such user inputs. If the formed symbolic PC cannot be solved (eg, infeasible path) or it is not formed yet (eg, when the editable widget is exercised at the first time), *smartCE* will generate a random user input. The random user input ensures that the exercising procedure proceeds without any interruption.

The *smartCE* engine is implemented in Java and uses the Symbolic Math Toolbox in MATLAB.[37] The Symbolic Math Toolbox provides a large set of functions for solving and manipulating symbolic math expressions. The symbolic math expressions can be solved either analytically or using variable-precision arithmetic. The *smartCE* communicates with the MATLAB symbolic engine by using third-party Java library, namely, *matlabcontrol*.[‡‡]

In Algorithm 6, we show the main steps of how *smartCE* generates a user input for the editable widget. First, *smartCE* sets the generated input (*in*) to empty (line 2). Next, *smartCE* extracts relevant input types from UI widget attributes such as "*text*," "*resource-id*," and "*content-desc*." It identifies the relevant input type for the editable UI widget *ew* (line 3) by scanning through the DBT2, which consists of the commonly used input types and their corresponding default or user-predefined input values. Based on the found input type *iType*, *smartCE* obtains the corresponding input value from DBT2 (line 5). If the input type is not found in DBT2, *smartCE* applies an input generation procedure (lines 7-15). If the formed *pc* is not empty (line 7), *smartCE* tries to generate a concrete user input *in* by solving the given *pc* (line 8).

[‡‡]https://www.cs.virginia.edu/~whitehouse/matlab/JavaMatlab.html

If no solution is found for the *pc* (line 9), *smartCE* tries to obtain a default value (eg, an actual value or hint) that is, by default, set by the app itself (line 10). If the default value does not exist (line 11), *smartCE* generates a random user input *in* (line 12). In the end, *smartCE* returns the generated user input *in* (line 17).

Algorithm 6: smartCE

```
   Input: pc – symbolic path constraint, ew – current editable widget
   Output: in – generated input for current editable widget
1  procedure smartCE(pc, ew)
2      in = < empty >;
3      iType = DBT2.GetType(ew);
4      if (iType.isFound()) then
5          in = DBT2.GetCustomValue(iType);
6      else
7          if (pc.isNotEmpty()) then
8              in = GenerateConcreteInput(pc);
9              if (in.isEmpty()) then
10                 in = ew.GetDefaultValue();
11                 if (in.isEmpty()) then
12                     in = GenerateRandomInput();
13                 end
14             end
15         end
16     end
17     return in;
18 end
```

3 | DESIGN AND IMPLEMENTATION OF MOBOLIC: PRACTICAL EXPERIENCE

In this section, we provide details of *Mobolic* practical experience that is learned during design and development phases. For that purpose, we establish several important questions to answer. The questions concern (1) the automated GUI traversing via the ***f-GFG*** model using the A* technique and (2) the automated generation of the relevant inputs via ***customated input generation*** using concrete, random, and user-predefined values.

3.1 | How do we implement GUI traversal technique using A* in practice?

In practice, *Mobolic* implements automated GUI traversing of the mobile apps using the existing A* technique. In fact, due to the complex nature of the mobile app GUI structure, it is not practical that modern app GUIs consist of only a single UI. Our observations show that a loop usually occurs during the exercising procedure for all but trivial app GUIs. Hence, if the loop has occurred, we need *Mobolic* to do traversing in *f-GFG* in order to return to the lastly discovered app UI that *Mobolic* has left during exercising. In particular, to find a path from an app UI to the lastly discovered one in the *f-GFG* model, *Mobolic* uses the UI widget execution trace. During the exercising procedure, *Mobolic* records the execution trace following the innate GUI execution path. Along the GUI execution path, *Mobolic* marks all the exercised UI widgets with flag "1" and memorizes the transitions from the exercised UI widgets to their subsequent app UIs. If during exercising, the current app UI on the device screen is changed to any previous one that is already added in *f-GFG*, *Mobolic* performs the UI search starting from that previous app UI. In particular, for every app UI, *Mobolic* recursively searches for the UI

widgets satisfying 2 conditions, where (1) the UI widgets are marked with flag "1" (ie, exercised) and (2) the UI widgets have memorized transitions leading to their subsequent app UIs. If such conditions are both satisfied, *Mobolic* performs a relevant action, eg, "*Click*," upon the corresponding UI widget to switch to its subsequent app UI according to the transition. *Mobolic* repeats the UI search until the lastly discovered app UI is reached and becomes the current app UI displaying on the device screen. Next, *Mobolic* continues the GUI exploration from the current app UI by exercising its retaining widgets.

3.2 | How do we relate editable UI widgets with Java code in practice?

To use the generated concrete user inputs, *Mobolic* needs to relate the editable UI widgets to the level of code where they are used. Having such relations, *Mobolic* knows in which particular editable UI widget the concrete user input should be injected. To resolve the problem of relating the code with the UI, *Mobolic* leverages the intermediate representations Jawa[§§] and smali, which are generated by Amandroid and Apktool, respectively. For example, we have a dice game *Yahtzee*. On its first UI, we need to indicate a number of rounds to play. Using the UI hierarchy viewer,[¶¶] a tool from Android SDK for GUI inspection, we can see that the editable UI widget relating to the number of rounds has a *resource-id* attribute with the value *com.tum.yahtzee:id/editText_rounds*, where *com.tum.yahtzee* is the app package name, and *editText_rounds* is the *id* of the UI widget. In fact, every Android app contains *ids* for all its UI widgets. Therefore, when *Mobolic* disassembles the app into smali using Apktool, it retrieves *ids* in human-readable form in the file *R$id.smali*. From the *resource-id* attribute, *Mobolic* uses the package name *com.tum.yahtzee* to construct the path " … */com/tum/yahtzee/*," where the file *R$id.smali* is located on disk. Next, *Mobolic* scans through *R$id.smali* and searches for *editText_rounds*. In particular, *Mobolic* performs search with a text that matches the string "*editText_rounds:I =* ." Once it has found the matching string, *Mobolic* retrieves an integer value that is represented in the hexadecimal system on the right of the symbol "=" and corresponds to *editText_rounds*, eg, 0x7f050010. Next, *Mobolic* uses Amandroid to disassemble the app and obtain Jawa. Note that in Jawa, UI widgets *ids* are represented in the decimal system; hence, *Mobolic* converts 0x7f050010 into the decimal value, ie, 2131034128. To find the name of the Java object that corresponds to the widget on the app UI, *Mobolic* scans through all the *.jawa* files and searches for the Jawa object with *id* 2131034128. Once the Jawa object is found, *Mobolic* searches for the nearest *AssignmentStatement*, and by using the data dependencies, it identifies that the Jawa object with *id* 2131034128 has the corresponding name *roundsText* in the Java source. As a result, by using the *ids* from smali and Jawa, *Mobolic* identifies that the Java object with the name *roundsText* relates to the editable UI widget *editText_rounds*.

3.3 | How do we generate customated user inputs in practice?

The customated input generation mechanism encompasses 3 types of the generated user inputs: *Random/Default*, *Concrete*, and *UI-context-aware*. For each input generation method, we set a priority. Thus, *UI-context-aware* is the first method to perform, *Concrete* is the second method to perform, and *Random/Default* is the last method to perform. Note that once any of the methods return the generated input, *Mobolic* stops the input generation process and uses the returned input in the exercising process. Defining such sequence, *Mobolic* finds the most relevant textual inputs to the current app UI. In the following sections, we describe each of the methods and explain how they are implemented in practice.

3.3.1 | UI-context-aware user input generation

Mobolic uses the *UI-context-aware* input generation method to obtain input values that lie in a set of possible values with a specific input pattern required, eg, email address or IP/URL of the server. All of such inputs may have different actual values; however, valid ones can only be the inputs with a correct pattern. For example, for the email address, it can be "*email@host.com*," and for the server address, it can be "*88.14.36.11*" or "*xxx00053.outlook.com*." These values have a predefined pattern that can only be recognized if a valid pattern of the email address or a valid pattern of the server address is used. Thus, if the email address is required by the app, the user should not provide any random characters without a specific pattern of the email, eg, "*aaa.email.com*" will not be recognized as a valid email so that the app is unlikely to proceed with such input even if the app does not do any input validation. Therefore, such invalid user inputs may block

[§§] http://pag.arguslab.org/jawa-language
[¶¶] https://developer.android.com/studio/profile/hierarchy-viewer.html\#start

the testing procedure, and thus, human interaction is required. Therefore, the *UI-context-aware* inputs are designed to resolve such cases and automate the testing process. In practice, we found that the most commonly used basic terms are "*password*," "*login*," "*email*," "*e-mail*," "*server*," "*port*," "*url*," "*phone*," "*account*," "*id*," "*name*," "*number*," "*tel*," and "*mobile*." For these text terms, we create a dictionary with the pair "*key-value*," where the terms serve as a "*key*" and the default values with a correct pattern serve as a "*value*." Note that our dictionary does not allow "*key-value*" pairs to be duplicated, ie, for the same "*key*," only one "*value*" is allowed at a time (ie, per testing app). For example, if the pair "*email = mobolic@gmail.com*," where "*email*" is the "*key*" and "*mobolic@gmail.com*" is the "*value*," has already been added to the dictionary, no other email address can be added, ie, only one record with "*email*" is allowed. As such, the dictionary will not contain multiple email addresses for one testing app. The same rule applies to other *keys* existing in the dictionary. In practice, our dictionary consists of the input types and their corresponding default valid input values. For example, for an editable UI widget that requires the *email* input type, the dictionary defines a default valid input value "*email@example.com*." However, optionally, to simulate a real-case scenario, the user may create an actual email, eg, "*mobolic@gmail.com*," and provide this email address to *Mobolic* by setting the field "*email*" in the "*config.properties*" file so that *Mobolic* will automatically replace the default email address by the user-provided one.

To summarize, we stress that *Mobolic* uses the *UI-context-aware* input generation method to automatically find which input type is required for a particular editable widget and to generate either predefined or user-provided inputs corresponding to the found input type. As such, the user does not need to be involved in the testing process. Instead, the user may provide actual values for the most common input types such as "*email*," "*password*," "*login*," "*server*," "*port*," "*url*," and "*phone*" in the "*config.properties*" file only once, before the exercising process starts. Thus, during exercising, *Mobolic* finds the required input type and injects its corresponding value into the relevant editable widget.

It is important to note that *Mobolic* allows the re-exercising of the same widgets multiple times on the previously encountered app UIs. For example, an app has widgets the traversing of which generates dependent app UIs, where every subsequent app UI depends on its previous one. Also, assume that at a certain point in the GUI, the dependent app UIs form a loop among themselves. Therefore, due to the loop existence in the GUI, to exercise all the widgets on the dependent app UIs, *Mobolic* has to revisit the app UIs several times. In turn, to re-visit the dependent app UIs, *Mobolic* has to traverse the same widgets multiple times. As such, *Mobolic* will generate the relevant user inputs and UI events on the respective app UIs as many times as needed, until all their widgets are exercised.

3.3.2 | Concrete user input generation

We implement the *Concrete* input generation method to enable *Mobolic* to generate inputs that should be an exact value or within an exact range. For example, if the app requires an "*age*" input and the input is validated in the code, eg, $(age > 0)\,\&\&\,(age < 16)$. For such scenario, the *Concrete* input generation method will generate the input value to satisfy the "*age*" condition. To extract $(age > 0)\,\&\&\,(age < 16)$, *Mobolic* performs the app code instrumentation by running its *appInster* component. The app code instrumentation is used to collect UID conditional statements. *Mobolic* uses these conditional statements to form the UID symbolic PCs. In turn, the UID symbolic PCs are to be solved by *Mobolic's* component *smartCE*, which is built on top of the MATLAB symbolic engine. Hence, in particular, when *smartCE* solves the UID symbolic PC $(age > 0)\,\&\&\,(age < 16)$, it returns a numeric value within the expected range (0,16), which will be defined by the underlying symbolic engine from MATLAB.

To summarize, we stress that *Mobolic* generates concrete user inputs using the dynamic symbolic execution technique. It collects symbolic constraints via app code instrumentation and forms possible symbolic PCs. *Mobolic* symbolically executes only UID conditional statements, whereas other statements are executed concretely. That is, at runtime, the other variables in the UID conditional statements take concrete runtime values, whereas symbolic values are to be found via solving their corresponding symbolic PCs.

Our implementation of the *Concrete* input generation method is different from that of the conventional one. It differs in that we execute symbolically only UID conditional statements, whereas the conventional method attempts to symbolically execute all possible conditional statements in the code. Using the *Concrete* input generation method, *Mobolic* aims to cover all UID conditional statements it has encountered on the execution path. For that purpose, *Mobolic* forms symbolic PCs for every discovered editable widget on the app UI and solves it using the MATLAB symbolic engine. In practice, along with the recorded symbolic PCs, *Mobolic* stores the generated concrete inputs so that they can be reused for the next exercising cycle. It is important to note that *Mobolic* continues generating the same user inputs for the corresponding editable widgets on the required app UIs until all their subsequent app UIs, depending on these inputs, are fully exercised.

Mobolic interacts with MATLAB using "*matlabcontrol*" library. First, *Mobolic* prepares a MATLAB script. It consists of algebraic equations and/or inequalities depending on the symbolic expressions in the formed PC. Next, *Mobolic* writes the prepared script into the "*solution.m*" file. Note that the formed symbolic path includes symbolic constraints concatenated with logical "&&" operator so that in MATLAB, the symbolic PCs can be represented as a system of algebraic equations and/or inequalities. As such, it guarantees that the system of algebraic equations and/or inequalities must be satisfied in order to have a satisfactory solution for the whole symbolic PC. Next, *Mobolic* uses "*matlabcontrol*" to send the script to the MATLAB symbolic engine for execution. Once the system is solved for every symbolic variable, MATLAB returns the solution by writing it into the "*solution.res*" file. It records the found values for every symbolic variable in the symbolic PC. Note that the solution can be returned in integer form (eg, $-1, 0, 5$, and others) or fractional form (eg, $\frac{3}{7}$, $\frac{2}{5}$, and others). The fractional-form values will be injected into the editable widget in decimal values with floating point, eg, $\frac{3}{7}$ as 0.42857142857 or $\frac{2}{5}$ as 0.4.

Mobolic uses MATLAB to solve the symbolic path constraints, the solutions of which lie in the numerical domain. However, there are cases when a textual solution is needed. For that purpose, we implemented a string resolver (in Java) to support 3 main text operations: "*equals*," "*compare*," and "*contains*." Thus, in order to find the textual input, we parse the expression into 2 parts and obtain the constant string value from either side. For example, *"name".equals(s)* or *s.equals("name")* are equivalent statements, where "*name*" is a string constant and "*s*" is a symbolic variable that belongs to the UID conditional statement. In order to find a value of "*s*," we extract the constant value from another side of "*s*," ie, from the leftmost or the rightmost position of the statement. For example, the code can be written as *s.toString().toLowerCase().equals("name")* or *"name".equals(s.toString().toLowerCase())*.

Our string resolver finds textual inputs assuming that "*equals*," "*compare*," and "*contains*" answer the following questions: for "*equals*": is it equal to/same as? – "Yes"; for "*compare*": is it more/less compare to? – "Yes"; for "*contains*": does it contain? – "Yes." Therefore, if an actual condition in Java is written with the meaning, eg, "NOT contain" or "NOT equal" (!*"name".contains(s)* or !*"name".equals(s)*), to satisfy the conditions, our string resolver will negate the found solution for "Yes" by providing the reversed string if the "Yes" solution is not empty or a random nonempty string if the "Yes" solution is an empty string. As such, the string resolver ensures that "*if*" and "*else*" logical branches, requiring textual solutions, are covered.

3.3.3 | Random and default user input generation

We allow *Random/Default* inputs to ensure that *Mobolic* does not require human interaction in the testing process. In case inputs are not found by *Concrete* or *UI-context-aware* methods, *Mobolic* will take default or random values (numeric or string). *Mobolic* first checks if the editable UI widget contains a default value (eg, an actual value or hint in the editable widget), which is provided by the app itself. If so, *Mobolic* memorizes this value for future input generation. If the editable widget is empty (no default value or hint), for the random input, *Mobolic* sets a predefined textual value "*defaultValue*" or generates a random integer value in a range of [1,100]. Depending on the app, if a numeric value is rejected (eg, only string values are allowed in the editable widget), *Mobolic* injects the predefined textual value.

4 | DEMO OF MOBOLIC

In this section, we show a workflow of *Mobolic* on the app whose GFG is shown in Figure 1. In this demo, we show how *Mobolic* explores the app GUI using the *f-GFG* model and how *Mobolic* generates user inputs via the customated mechanism. Note that, in this demo, we focus on the symbolic part of a customated input generation mechanism due to the complexity of demonstrating a mixture of all possibly generated inputs. For more details about *UI-context-aware* and *Random/Default* user input generation, refer to our conference paper.[22]

In our demo, we assume that the app consists of 3 UIs. On u_1, the app has an editable widget e_{11}. The injected input for e_{11} is passed as an input parameter to the function "*foo*" whose code and r-CDG are shown in Figures 2 and 3, respectively. Note that, in Figure 2, *code#4* in line 15 does not depend on the user input. As such, it may be executed or not depending on the system time, which is obtained at that moment when the function "*foo*" is invoked and the conditional statement in line 14 is executed. The function "*foo*" is always invoked when a relevant UI event is fired upon the noneditable widget w_{12}. Without loss of generality, we assume that the app code does not have any other UID conditional statements except those in the function "*foo*."

Step#1. First, *Mobolic* executes *stAnalyzer*, which builds r-CDG as shown in Figure 3. From Figure 3, we can see that the function "*foo*" has 3 execution paths $1 \Rightarrow 2$, $1 \Rightarrow 4 \Rightarrow 5$, and $1 \Rightarrow 4 \Rightarrow 6$ as well as 2 decision nodes (1) and (4), which are constrained by the user input from the editable widget e_{11}. Next, guided by r-CDG, *Mobolic* executes *appInster* to instrument the app. *Mobolic* sets a set of fully exercised UIs *feui* and *f-GFG* to be empty.

Step#2. Next, *Mobolic* runs the instrumented app and obtains the first UI u_1. Next, *Mobolic* executes *iScheduler* and *smartCE* to generate a relevant user input. Since the editable widget e_{11} is exercised for the very first time, *smartCE* generates a random user input, eg, 11. *Mobolic* injects the input into e_{11}.

Step#3. Afterward, *Mobolic* executes *SAT* to exercise noneditable widgets, build *f-GFG*, and collect UID symbolic constraints through app instrumentation. *SAT* adds u_1 with its widgets into *f-GFG* and finds the first unexercised relevant noneditable widget on u_1, ie, w_{12}. *SAT* exercises w_{12} by generating a relevant UI event for w_{12}, eg, "*Click*" event. Upon exercising w_{12}, *Mobolic* invokes the function "*foo*" so that *code #1* is executed since the UID conditional statement $(x > 5)$ corresponding to the decision node (1) is satisfied by the random input, ie, 11, which has been generated in Step#2. Also, after clicking on w_{12}, u_2 is discovered. *SAT* stores the corresponding symbolic constraint $[(x > 5)]$ that is encountered on the execution path $1 \Rightarrow 2$ and the generated input 11. Also, it adds the link $w_{12} \rightarrow u_2$ and u_2 with its widgets into *f-GFG* and returns *infloop* FALSE to indicate to *Mobolic* that there is no loop occurrence in the exercising procedure. As such, the current UI is now u_2, and the procedure continues.

Step#4. *Mobolic* repeats Step#3 for u_2 and u_3. After exercising w_{31} on u_3, the current UI is u_1. Since u_1 is already in *f-GFG*, the link $w_{31} \rightarrow u_1$ will form a loop in *f-GFG*. Therefore, to eliminate the loop, *SAT* will not add the link $w_{31} \rightarrow u_1$ into *f-GFG*. Instead, *SAT* returns *infloop* TRUE to indicate to *Mobolic* that there is a loop occurrence in the exercising procedure. Afterward, *Mobolic* performs A* search in *f-GFG* to find the shortest path from the current UI u_1 to the lastly discovered UI u_3. The found path includes $w_{12} \rightarrow u_2; w_{21} \rightarrow u_3$. Since u_3 is constrained by the user input from the editable widget e_{11}, *iScheduler* retrieves the previously generated user input from e_{11}, ie, 11. *Mobolic* injects the input, replays the found path, and reaches the lastly discovered UI u_3. Therefore, u_3 becomes the current UI, ie, UI that is currently displaying on the Android device screen.

Step#5. On the current u_3, *Mobolic* finds the next unexercised noneditable widget, ie, w_{32}. *SAT* generates a relevant UI event for w_{32} , eg, "*Click*" event. However, exercising w_{32} leads to the same UI, ie, u_3, and thus, this transition forms a loop in the exercising procedure. Since u_3 is already in *f-GFG*, the link $w_{32} \rightarrow u_3$ will not be added into *f-GFG*. *SAT* returns *infloop* TRUE to indicate to *Mobolic* that there is a loop occurrence in the exercising procedure.

Step#6. After exercising w_{32}, *Mobolic* realizes that the UIs u_2 and u_3 are fully exercised. As such, *Mobolic* adds u_2 and u_3 with all their widgets into the set of fully exercised UIs *feui* and removes them from *f-GFG*. As a result, in *f-GFG*, *Mobolic* changes the lastly discovered UI to u_1. Also, *Mobolic* realizes that the current UI displaying on the device is still u_3, which is not already in *f-GFG*. Therefore, *Mobolic* generates a system event "*Back*." Note that the "*Back*" event brings the app to any previously discovered UI in *f-GFG*. Without loss of generality, we assume that the event brings the app UI to u_1. Thus, after the "*Back*" event, u_1 becomes the current UI.

Step#7. Even though all noneditable widgets are fully exercised on all the discovered UIs u_1, u_2, and u_3, *Mobolic* continues the exercising procedure since the decision nodes (1) and (4) are not fully covered yet. From the stored symbolic constraints for e_{11}, *iScheduler* forms a symbolic path constraint *pc*, ie, $\{(x > 5)\}$. However, *iScheduler* realizes that $pc\{(x > 5)\}$ has already been covered by input 11 that has been generated in Step#3. As such, it negates the $pc\{(x > 5)\}$ and forms another one, ie, $\{(x \leq 5)\}$.

Step#8. The *smartCE* generates a user input by solving the formed $pc\{(x \leq 5)\}$, eg, it generates a user input equal to 4. *Mobolic* injects the input into e_{11} and updates the corresponding symbolic constraints for e_{11} from the formed symbolic path constraint $pc\{(x \leq 5)\}$. Hence, the updated symbolic constraint for e_{11} is now $[(x \leq 5)]$.

Step#9. Next, *Mobolic* executes *SAT* to exercise w_{12}. Upon exercising w_{12}, *Mobolic* covers the execution path $1 \Rightarrow 4 \Rightarrow 6$ and executes *code#3* since the UID conditional statement $(x = 5)$ in the decision node (4) is not satisfied, ie, $x = 4 \neq 5$. For the editable widget e_{11}, *SAT* stores the generated user input 4 and the newly discovered symbolic constraint $[(x = 5)]$, which is encountered on the execution path $1 \Rightarrow 4 \Rightarrow 6$. Thus, the editable widget e_{11} has now 2 new symbolic constraints $[(x \leq 5)]$ and $[(x = 5)]$. Note that the current UI u_1 has not been changed since only the user inputs, satisfying the condition $x > 5$, lead to discovering new app UIs u_2 and u_3, while other user inputs do not change u_1 (see Figure 2).

Step#10. Next, *iScheduler* forms a symbolic path constraint $pc\{(x \leq 5)\ \&\&\ (x = 5)\}$ from the stored symbolic constraints for e_{11}. The *smartCE* generates a user input by solving the formed $pc\{(x \leq 5)\ \&\&\ (x = 5)\}$. As a solution, it generates a user input equal to 5. Next, *Mobolic* injects the generated input into e_{11} and executes *SAT* to exercise w_{12}. Since the UID

TABLE 2 Summary of characteristics of the 10 apps tested

App Category	App Name	App Version	LOC	SGUI	WGUI
Games	Guess[a]	0.12	46	3	4
	Hangman[a]	0.3.1-alpha	142	4	4
	Yahtzee[a]	1.1	496	14	51
	Cowsay[b]	1.4	243	55	183
	GM Dice[b]	0.1.6	560	160	259
	MunchLife[b]	1.4.3	161	5	16
Utility	Pedometer[b]	1.4.1	985	39	96
	Ringtone generator[b]	0.4	694	11	40
	Bodha Converter[a]	1.0	779	49	180
	Authenticator[a]	2.21	1937	19	50

[a]Apps requiring specific user text inputs. [b]Apps accepting random user text inputs.

conditional statement ($x = 5$) in the decision node (4) is satisfied, ie, $x = 5 = 5$, *Mobolic* covers the execution path $1 \Rightarrow 4 \Rightarrow 5$ and executes *code#2*. The current UI u_1 is not changed. Note that *Mobolic* does not update the corresponding symbolic constraints for e_{11} from the formed symbolic path constraint $pc\{(x \leq 5)\ \&\&\ (x = 5)\}$ since e_{11} already includes the same symbolic constraints from Step#9.

Step#11. Next, *Mobolic* realizes that the decision nodes (1) and (4) are fully covered since all their corresponding UID conditional statements with all their subconditional statements are executed, and thus, the current UI u_1 is fully exercised. Hence, *Mobolic* adds it with all its widgets into the set of fully exercised UIs *feui* and removes it from *f-GFG*. At this moment, *Mobolic* realizes that all discovered UIs u_1, u_2, and u_3 are fully exercised, and thus, *Mobolic* terminates.

5 | EMPIRICAL EVALUATION

Mobolic is designed to be a practical approach; therefore, it allows the automated testing procedure to notably shorten the overall exercising time and increase code coverage. To demonstrate *Mobolic* performance in practice, we downloaded 10 real-world open-source Android apps from F-Droid,[##] performed testing, and compared results with several state-of-the-art automated GUI-testing approaches. The obtained results have provided first evidence of *Mobolic* practicality (see Section 5.6). Of the 10 apps, 5 are with primitive and 5 are with nontrivial GUI structures whose code is UI related; we also have 5 apps that require specific user inputs and 5 apps that may accept random user inputs. We selected apps with particular properties (complexity of GUI and user inputs) that could be sufficient to show adequacy and feasibility of our approach.

To evaluate *Mobolic* performance, we ran it on 10 downloaded apps and compared the obtained code coverage results and exercising time with other prevalent approaches, including manual testing (Human) and 3 automated ones (Sapienz,[13] ACTEve,[8] and MobiGUITAR[7]). We chose Human for comparison since manual testing can provide the most intelligent inputs. We chose Sapienz as a representative of the multiobjective search-based testing, which shows the best performance compared with Monkey and Dynodroid tools involving a random GUI exploration strategy. Note that Sapienz extends the model-based testing by employing the search-based approach and exploiting an automated random user input generation. We compared with ACTEve since it is the only tool that performs automated concolic testing. Also, we compared with MobiGUITAR as a representative that shows the best performance in comparison to A^3E and *SwiftHand* tools involving a systematic GUI exploration strategy.

5.1 | Experimental data set

In this section, we provide the summary of characteristics of the 10 selected apps showing in Table 2. We provide the "*category*," "*name*," and "*version*" of the apps used in our experiment. In addition, we provide app characteristics such as "LOC" (total number of *executable lines* of the Java code), "SGUI" (total number of *screens* in the app GUI), and "WGUI" (total number of *accessible widgets* for all discovered screens in the app GUI).

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

TABLE 3 Android emulator device specifications

Parameter	Value
Device	Galaxy Nexus
Target	Google APIs (API Level 21)
CPU/ABI	Intel Atom (x86_64)
Hardware	Keyboard Present
RAM	2 GB
VM heap	128 MB
Front/Back cameras	None
Internal storage	500 MB (with default content)
SD card	200 MB (with default content)

The "SGUI" value consists of the summation of every rendered screen (ie, app UI hierarchy view) discovered in the app GUI as guided by the transitions existing between the screens. Note that "SGUI" may include duplicated screens since the app GUI may have transitions leading to the same screen multiple times. The "WGUI" value consists of the summation of every accessible widget found on every discovered screen in the app GUI. We define an accessible widget as the widget that can be exercised via the app GUI by triggering any of the events supported by *Mobolic* (see Table 1). The value "LOC" is obtained via the EMMA tool (see Section 5.3.5), and values "SGUI" and "WGUI" are obtained via *Mobolic*.

From Table 2, we can see that the selected apps fall into 2 categories "Games" and "Utility." To give a sense of what each app is about, we describe each as follows.

1. *Guess*: Our aim as a user is to guess a hidden random number within the [1,100] range with 8 trials.
2. *Hangman*: The app randomly picks 1 of 40 000 English words from its database. Our aim as a user is to guess the hidden word. The app allows the user to type only a single character at a time.
3. *Yahtzee*: The app is a dice game. Generally, the game consists of a number of rounds and involves multiple players. To initiate the game, it should consist of at least one round and involve one player.
4. *Cowsay*: It turns text into ASCII cows, with speech (or thought) balloons.
5. *GM Dice*: It is a dice rolling application with special focus on the 3*D*20 role-playing game system.
6. *MunchLife*: It is a simple counter application for keeping track of your level while you are playing the card game Munchkin.
7. *Pedometer*: It is a step counter with speed, distance, steps-per-minute, and text-to-speech.
8. *Ringtone generator*: It generates Morse code ringtones in WAVE and iMelody formats.
9. *Bodha Converter*: The app supports general (binary, octal, decimal, and hex) conversions up to $(2^{63}-1)$ in their respective formats. It also supports ASCII character conversion in the range of NUL to DEL, ie, (0−127).
10. *Authenticator*: The app implements one-time password generators. When generating the one-time password, it requires that the user key should not be shorter than a certain length.

5.2 | Experimental environment

To run *Mobolic*, we use Android emulators on a Linux local host. The Linux machine was running 64-bit Ubuntu 14 on a 4-core CPU-i5 with 16 GB of RAM. For the development of *Mobolic*, we use the Java programming language (Java 7) and the *UI Automator* library from the Android SDK. To communicate with the Android emulators, we use the Android Debug Bridge (*adb*) tool.

For our experiment and evaluation, we use Android emulators as this is the traditional and cost-effective solution. In addition, the Android emulators can be freely accessed and is suitable for performing UI, stress, and performance testing. However, *Mobolic* can also be deployed on the physical Android devices since this does not require any instrumentation of the Android framework. By manual checking, we ensured that all our selected apps do not require any specific Android device hardware so that our test results are not affected by running the apps on the Android emulator. As such, we configured the emulator as shown in Table 3, and all other settings retained their default values. Also, we did not modify the Android platform, and instead, we used all its default system images.

5.3 | Experiment design

In this section, we describe a design of our experiment following the recommended guidelines.[38,39] Here, we state the *rationale* and *objectives* of this study as well as establish several *Research Questions (RQs)* to answer, define *concepts and measures* and the *methods of data generation and collection* used in our study, and identify the *data selection strategy*.

5.3.1 | Rationale: why this research work has been done?

We have been investigating previous research works about the automated testing of the GUI of mobile apps. From our investigations, we found that they were trying to address the issue with poor code coverage while designing their approaches to be fully automated. However, eventually, the developed approaches have managed to achieve, on average, around 45%-50%. As we found, the main obstacles to achieving high code coverage (in practice, achieving 85%-90%, on average, is usually a good coverage) are (1) *automated input generation* and (2) *automated GUI exploration strategy* since manual human interaction is to be avoided. The previous works were trying to tackle these 2 main issues by applying different techniques to automate their approaches, eg, designing novel automated input generation systems, using dynamic symbolic execution for input generation, improving the random GUI testing technique, and developing different applications of the model-based technique for systematic GUI exploration. However, upon publishing the works, the developed approaches did not demonstrate in a convincing way that the issues were indeed resolved, or at least addressed to a certain extent via a notable improvement.

We found that the previous research works do not address those 2 issues simultaneously. Instead, they focus on improving either the automated input generation or the GUI exploration strategy. However, even proposing different solutions to address either of the 2 issues, the developed approaches were found to be not much practical. For example, in practice, many of the developed approaches usually require an astronomical amount of time for app exercising, or automatically generated textual inputs are not sensible enough for proper systematic GUI exploration. As such, to develop automated approaches, the previous research works rely on the user's availability to manually set, eg, a termination condition for their testing tools since they are not able to stop the testing process automatically or a certain limit for the depth exploration of the app GUI to mitigate the problem with exponentially growing exercising time, or suggest for the user to manually provide sensible textual inputs during the testing procedure or before the procedure is started for the purpose of automatic replaying the user inputs.

From our investigations, we can see that the issues with poor code coverage and inadequate automated input generation still remain, such that many other research works have been further conducted to propose alternative automated approaches. Therefore, we believe that there is still great relevance and necessity in providing a solution that addresses both issues. As such, in our work, we propose an automated GUI testing approach, namely, *Mobolic*, that introduces (1) a novel ***customated input generation*** mechanism to eliminate human interaction in the testing process and (2) a novel ***f-GFG*** model of the app GUI to efficiently and effectively perform systematic GUI exploration in an automated manner. Therefore, these 2 novel aspects enable our approach to significantly increase code coverage results and shorten the exercising time, which makes our approach more practical.

5.3.2 | Objectives: what is expected to be achieved with this research work?

In this research work, we raise several objectives to achieve. We list them as follows.

1. We are to develop an automated approach that allows, for mobile apps, to automatically generate functional UI tests with high coverage within a reasonable amount of time via the systematic GUI exploration. We expect that our proposed approach achieves high code coverage and significantly reduces exercising time (in comparison with existing approaches).
2. We are to develop an automated approach that allows, for mobile apps, to automatically generate sensible user inputs that are relevant to the current app UI. We expect that our proposed approach is able to find relevant inputs for many real-case scenarios so that human interaction is not required.
3. We are to perform a comparison of our approach with state-of-the-art existing automated approaches and a manual one. We expect that our proposed approach, on average, outperforms the existing automated approaches and the manual one, while showing higher code coverage results and shorter exercising time.

5.3.3 | Research Questions: what knowledge will be sought or expected to be discovered?

In this work, we establish several *RQs* to answer. We list them below as follows.

RQ#1. How does our *Mobolic* testing technique impact a code coverage in comparison with the manual testing of the mobile app GUI?
RQ#2. How does our *Mobolic* testing technique impact a code coverage in comparison with the random testing of the mobile app GUI?
RQ#3. How does our *Mobolic* testing technique impact a code coverage in comparison with the dynamic symbolic execution of the mobile app GUI?
RQ#4. How does our *Mobolic* testing technique impact a code coverage in comparison with the model-based testing of the mobile app GUI?
RQ#5. How does our *Mobolic* testing technique impact an exercising time in comparison with the state-of-the-art existing automated testing techniques of the mobile app GUI?

To answer these *RQs*, we include Section 5.6 where we do analysis and interpretation of the experimental results.

5.3.4 | Concepts and measures: which coverage metric (measure) for executed Java code do we use?

Code coverage is the most useful indicator showing testing approach effectiveness, simply because the lower the coverage results, the more app code that remains untested. The EMMA suggests that "*basic block*" is a fundamental coverage metric, and other coverage metrics such as "*branch*," "*class*," "*method*," and "*line*" can be derived from *basic block*." As such, in our experiment, we use the "*basic block*" coverage metric for the evaluation of the code coverage results. In fact, "*basic block*" is a well-defined metric, and it is represented as a sequence of bytecode instructions without any jumps or jump targets inside, ie, an atomic unit.

EMMA marks a "*basic block*" as covered once the control has reached its last instruction. Therefore, the "*basic block*" is guaranteed to have been executed without failure at least once and reflects more precisely how adequately the app functionality was exercised, rather than "*lines*," "*classes*," or "*methods*." Note that the "*line*" coverage metric is a poorly defined concept and should not be used as a code coverage metric except when linking line coverage to the original source code. In contrast, the "*class*" and "*method*" metrics are well defined but indeed cannot be used to show how adequately the app functionality was tested. In particular, EMMA marks a Java "*class*" or "*method*" as covered once the class is loaded by the Dalvik VM or Java VM (depending on the running virtual machine), or method is entered (very first "*basic block*" of the method is covered). Instead, the "*class*" and "*method*" coverage metrics should be used to detect, eg, "*dead code*" in the app, rather than showing the performance in terms of achieved code coverage.

5.3.5 | Method of data generation: what instruments do we use for Java code coverage generation and collection?

For code coverage generation and collection, we use the EMMA tool.[11] EMMA is an open-source tool for measuring and reporting Java code coverage, and it is included in the Android SDK by default. EMMA can instrument Java bytecode (*.*class*) for coverage either offline (before *.*class* files loaded) or on the fly (when Java bytecode is being executed). EMMA supports coverage types such as "*class*," "*method*," "*basic block*," and "*line*." Note that "*line*" coverage is not provided without the source code being available. In addition, EMMA supports 3 types of coverage reports in *.*txt*, *.*html*, and *.*xml* file formats.

There are 2 ways to use EMMA with Android apps: (1) to include EMMA (pre-dexed form of *emma.jar*) into the Android system image (*system.img*) or (2) to include EMMA into *classes.dex* of the app (*.*apk*). We chose (2), which is an easier and more universal way and allows us to run apps on multiple Android platforms (if needed). Note that the pre-dexed form of *emma.jar* is not enough to make EMMA work on the Android platform. EMMA's folder structure with their content must also be included into the root of the testing *.*apk* file (the same folder level as *classes.dex*).

[11] http://emma.sourceforge.net/faq.html

5.3.6 | Method of data collection: how do we collect Java code coverage data?

To dump (collect) code coverage, we send an activity broadcast intent[***] to the app under test. To have unique intent names for each app, we use their package names appended with "*.EMMA_DUMP_COVERAGE*." Once such an intent has been sent, it calls the *onReceive(…)* method from our Java class, namely, "NxAe85AOZo9UAYV," for reasons of uniqueness, which extends the *BroadcastReceiver* Java class. The Java class "NxAe85AOZo9UAYV" is included into the *classes.dex* of testing **.apk* where we dump the coverage by calling EMMA's method *dumpCoverageData(…)*. Note that since we include EMMA in **.apk*, we are able to call EMMA methods directly without applying a Java reflection mechanism.

To let the app under test react to the sent broadcast intent, there are 2 ways: (1) to statically register a broadcast receiver in the *AndroidManifest.xml* file or (2) to dynamically register a broadcast receiver at runtime. We chose method (1) as it is a more reliable and less error-prone method. Note that in method (1), all receivers are registered globally, in contrast to method (2) where receivers can be registered globally or locally. If method (2) is selected, in order to register a receiver globally, the app context should be used, ie, *getApplicationContext().registerReceiver(…)*. We use method (1) to register the broadcast receiver globally since code coverage results are needed for the entire app (from all possible app states). If the broadcast receiver is registered locally (eg, for the specific app activity), coverage results can be collected only for the activity (**.class*) while it is running in the foreground.

To make sure that all coverage data are collected, after every performed action on the UI, we dump intermediate code coverage (*coverage.ec*) into the app under test folder on the emulator, ie, "*/data/data/app_package_name/files*." We use the default app's folder on the emulator to avoid making other changes to the *AndroidManifest.xml* file (eg, adding extra permissions to allow the app to write code coverage files on external storage). Note that coverage data will be lost if the app is crashed; therefore, an intermediate code coverage is necessary to obtain coverage results for any previously executed code.

5.3.7 | Data selection strategy: how do we identify executed Java code?

We identify the executed Java code via app code instrumentation using the EMMA tool. To instrument app **.class* files, we use the offline EMMA method. First, we generate Android **.apk* files from the source code using the Eclipse IDE for Android. For instrumentation, we consider only **.class* files that correspond to the source code **.java* files. However, we exclude autogenerated **.class* files such as "*BuildConfig.class*" and all "*R*.class*" files from EMMA instrumentation since they would contribute to the code coverage with "0" coverage values for all coverage metrics since they are not executable. Once all app **.class* files are instrumented, EMMA generates one "*coverage.em*" file with all the metadata required for code coverage report generation.

5.4 | Threats to internal and external validity

In our experiment, we identify 3 main *Threats (THs)* to the internal and external validity[39-41] of the obtained results for the *code coverage* and *exercising time*. We describe threats in the list below and propose solutions to mitigate or eliminate them (if possible).

TH#1 As a threat to the internal validity, it concerns about *code instrumentation* and *code coverage collection*. Due to the human error, in the app, there is a possibility that it has places with unreachable code, ie, "dead code," or, somewhere, it contains "unused code," eg, for future implementation, or forgotten to delete. Also, the app may implement "version-dependent" code via conditional statements to support multiple versions of the Android platform, or the app code can be "device dependent," ie, the runtime behavior on the Android emulator and the actual device is different as the app may require specific physical device hardware components. Hence, if we instrument the Java bytecode without prior checks for the code mentioned above, the EMMA tool will include such irrelevant code in the overall code coverage. For example, instead of the overall number of executable lines of the Java code, which is 1800*LOC* (expected, ie, true value), EMMA will report 2100*LOC* (misleading, ie, false value); hence, 300*LOC* is the extra code that cannot be executed as explained above. Thus, such threat may lead to unexpected code coverage results and, potentially, to misleading conclusions about the performance of *Mobolic*. Thus, to exclude such a code from instrumentation and code coverage, in our experiment, we check the source code for

[***] https://developer.android.com/guide/components/broadcasts.html

all the abovementioned code with the aid of the Eclipse IDE for Android. As such, we ensure that the obtained code coverage results are calculated using the expected true values of *LOC*.

TH#2 As a threat to the internal validity, it concerns about *completeness of GUI exploration* and *obtained code coverage*. In our experiment, we expect that *Mobolic* discovers as many app UIs as possible in the existing app GUI model. However, there is still a chance that not all possible app UIs are discovered. It is due to the fact that the app may not have entered its certain states that depend on the internal app logic implementation or textual user inputs that are provided during the testing process. As such, certain app UIs still may not be discovered. As a result, our code coverage results may not reflect an absolute code coverage value that is theoretically possible to achieve in an ideal case. However, such threat is uncontrollable due to the nature of the performed testing, ie, "black-box," where *Mobolic* does not have prior knowledge about the number of possible app UIs existing in the app GUI, which, theoretically, could be discovered.

TH#3 As a threat to the external validity, it concerns about *execution environment*, *experimental data set size*, and *exercising time*. The exercising time depends not only on the absolute execution time of the automated approaches but also on the execution environment. To eliminate the factor of execution environment, we run the testing tools on the same personal desktop computer. However, such external factors as tools configuration (eg, emulator parameters, testing tool parameters, and others), internal implementation (eg, it depends on the developer coding skills and code optimization), and testing techniques involved (eg, random, dynamic symbolic, search-based, or model-based testing techniques) impact the exercising time. In addition, for certain apps, the exercising time may be dramatically increased depending on the app itself (eg, complexity of the app GUI, required user inputs, or internal execution flows in the app). Even though the average exercising time for our experimental data set may not be used "as is" to rely upon in general, it nevertheless gives reasonable sensing of how efficient the testing techniques could be.

5.5 | Experiment operation: preparation and execution

In this section, we explain how we set up experiments for Human, Sapienz, ACTEve, MobiGUITAR, and *Mobolic*. We ran *Mobolic*, Sapienz, ACTEve, and MobiGUITAR for all 10 apps on 5 Android emulators and averaged the obtained code coverage results. For *Mobolic*, we did not set any time limit for the exercising procedure since it implements the self-terminative GUI exploration strategy. For Sapienz, ACTEve, and MobiGUITAR, we configured them using the suggested configuration,[†††] which was proposed by Choudhary et al.[14] As suggested, we also set a 60-minute exercising time limit for each app for Sapienz, ACTEve, and MobiGUITAR.

For Human, we selected 5 advanced Android users, including 2 of the authors of this paper, and students. We ensured that the selected students have knowledge in the testing of the mobile app GUIs and know their intricacies. We manually tested all 10 apps following rules such as (1) run the testing apps on the same Android platform as *Mobolic*, (2) avoid installing any dependent apps if the testing app prompts, and (3) navigate back if the testing app goes to any external app. After testing is completed, we averaged the obtained code coverage results. To make a fair comparison of Human with *Mobolic*, we did not set any time limit for the manual exercising since it innately cannot be extremely long due to the human nature but reasonable for every particular app. Thus, according to our reasoning about the app GUI and its depth exploration, we exercised each app until it was visually possible to identify the newly discovered app UIs.

5.6 | Experimental results: analysis and interpretation

We evaluate the effectiveness of Human, Sapienz, ACTEve, MobiGUITAR, and *Mobolic* in terms of achieved code coverage and exercising time. In Table 4, we show their obtained code coverage (columns 2-6) and exercising time (column 7) for *Mobolic*. Note that we do not report the exercising time for Human, Sapienz, ACTEve, and MobiGUITAR. In our experiment, we focus on the comparison of the effectiveness of *Mobolic* with Human, Sapienz, ACTEve, and MobiGUITAR in terms of the achieved code coverage. As such, for these approaches, we only report the code coverage. Since the exercising time is not reported for Human, Sapienz, ACTEve, and MobiGUITAR, we refer to Section 5.5 where we discuss how the exercising time is chosen. However, for *Mobolic*, we still report the exercising time for evidence purposes to show that it does not take an astronomically long time to exercise the apps.

[†††] http://bear.cc.gatech.edu/~shauvik/androtest/

TABLE 4 Code coverage for *Mobolic*, Human, Sapienz, ACTEve, and MobiGUITAR and the time taken by *Mobolic*

App Name	Mobolic, %	Human, %	Sapienz, %	ACTEve, %	MobiGUITAR, %	Mobolic, *mins*
Guess[a]	96	36	32	24	26	3.47
Hangman[a]	100	58	35	21	19	8.29
Yahtzee[a]	99	99	44	19	35	6.48
Bodha Converter[a]	98	98	53	16	48	39.81
Authenticator[a]	83	88	41	28	33	5.95
Cowsay[b]	91	80	78	43	47	49.47
GM Dice[b]	87	78	56	41	51	58.44
MunchLife[b]	89	89	58	42	49	5.08
Pedometer[b]	86	81	77	36	58	19.13
Ringtone generator[b]	93	77	60	49	69	23.95
Average	**92**	**78**	**53**	**32**	**44**	**22**

[a]Apps requiring specific user text inputs. [b]Apps accepting random user text inputs.

In Table 4, we show the code coverage, computed as follows:

$$\text{Coverage} = \frac{\text{Average number of covered basic blocks}}{\text{Total number of basic blocks}}.$$

As for Human, the average number of covered basic blocks is computed by taking the average of the results given by 5 users. As for *Mobolic*, Sapienz, ACTEve, and MobiGUITAR, the average number of covered basic blocks is computed by taking the average of the coverage results given by 5 Android emulators.

5.6.1 | To answer RQ#1: Mobolic vs Human (manual testing)

From Table 4, we can see that, on average, Human achieves lower code coverage with 78% as compared with *Mobolic* with 92%. In comparison with *Mobolic*, Human provides relatively low code coverage for only 2 apps "*Guess*" and "*Hangman*" due to their specific app nature. In particular, the app "*Guess*" prompts the user to guess a hidden random number within the [1,100] range with 8 trials. Similarly, the app "*Hangman*" prompts the user to guess the hidden word that is randomly chosen from the app database of 40 000 English words. Due to such specific app nature and the human reasoning behind them, it can be seen that Human will hardly guess the number or word correctly within the limited number of trials. Therefore, Human will likely be covering the code along the execution path that always loses the game and may never cover the code along the execution path winning the game. In contrast, through app instrumentation, *Mobolic* obtains the concrete value that is chosen by the app for the user to guess. Therefore, knowing that concrete value, *Mobolic* is capable of covering the code along both execution paths winning and losing the game. For the remaining apps, *Mobolic* and Human perform similarly.

5.6.2 | To answer RQ#2: Mobolic vs Sapienz (search-based testing)

From Table 4, we can see that, on average, Sapienz and *Mobolic* cover 53% and 92% of the code, respectively. In comparison with *Mobolic*, Sapienz provides relatively low code coverage. For "*Guess*" and "*Hangman*" apps, Sapienz achieves low coverage. Due to the underlying random-based input seeding mechanism, Sapienz is unable to exercise these 2 apps adequately since they require concrete user inputs. For "*Yahtzee*," "*Bodha Converter*," and "*Authenticator*" apps, Sapienz gives a higher coverage. However, these apps have complex GUI models and, thus, would require systematic GUI exploration for achieving a higher code coverage (in fact, it depends on the actual tool implementation). Therefore, for these apps, due to the underlying random-based GUI exploration strategy, Sapienz fails to provide high coverage as well.

For the other apps marked with symbol "b" in Table 4, Sapienz achieves its highest coverage. The results are simply explained by the fact that these apps are good to exercise with any random inputs since they accept any user inputs. However, we realized that Sapienz obtains the lowest coverage for "*MunchLife*" and "*GM Dice*" apps. After manual investigation, we found that the "*MunchLife*" app requires more complex UI actions such as "*Roll*" and "*Press*" to trigger particular execution paths covered. Similarly, for the "*GM Dice*" app, Sapienz obtains low coverage due to the lack of support for particular UI actions such as "*Scroll*" and "*LongClick*." In contrast, *Mobolic* supports the most comprehensive list of UI events and a limited set of systems (see Table 1), and its improved systematic GUI exploration strategy with custom-

ated input generation could discover certain execution paths that are inaccessible to the random-based GUI exploration techniques with random-input generation.

5.6.3 | To answer RQ#3: Mobolic vs ACTEve (dynamic symbolic execution)

From Table 4, we can see that, on average, ACTEve and *Mobolic* cover 32% and 92% of the code, respectively. In comparison with *Mobolic*, ACTEve provides significantly lower code coverage. We manually studied the nature of the underlying implementation of ACTEve to understand why the code coverage is low. We found that ACTEve explores the apps starting from their entry point and employs symbolic execution for the entire app rather than for particular statements. In sharp contrast, *Mobolic* performs symbolic execution on only UID statements rather than trying to symbolically execute all possible statements in the entire app. Therefore, in comparison with *Mobolic*, ACTEve may require a large amount of time to adequately exercise the app and achieve high code coverage. As a result, in practice, achieving high code coverage for ACTEve for all but trivial apps is generally impossible within a reasonable time limit, and thus, ACTEve is a time-capped approach. Also, ACTEve does not scale beyond event sequences consisting of more than four events, and thus, ACTEve is also a depth-limited approach. In sharp contrast, *Mobolic* does not have such limitations and forms event sequences of any length to reach a particular UID conditional statement and does so within a reasonable time.

5.6.4 | To answer RQ#4: Mobolic vs MobiGUITAR (model-based testing)

From Table 4, we can see that in comparison with *Mobolic*, MobiGUITAR provides significantly lower code coverage. On average, MobiGUITAR and *Mobolic* cover 44% and 92% of the code, respectively. In particular, MobiGUITAR provides the lowest code coverage for "*Guess*" and "*Hangman*" due to their specific app nature, where the apps prompt the user to guess a hidden random number or word, respectively, within several trials. However, MobiGUITAR is not able to generate concrete user inputs, and thus, it may never cover the execution paths that are constrained by concrete user inputs. For the rest of the apps, MobiGUITAR shows better performance since the app functionality does not mainly depend on the concrete user inputs, ie, random user inputs can also be generated to discover unexplored app UIs.

In fact, MobiGUITAR has an option to provide concrete user inputs manually through the tool's configuration file. However, in practice, it may require major manual efforts since the provided user inputs will be relevant to only the specified app (due to MobiGUITAR implementation); therefore, the provided inputs cannot be reused by another app. That is, if the inputs are to be provided by a user, it will require the user to manually specify inputs for every app to be tested. In contrast, *Mobolic* is designed to automatically generate relevant user inputs through the customated input generation mechanism so that the constrained app functionality can be adequately exercised without human intervention.

5.6.5 | To answer RQ#5: Mobolic vs state-of-the-art automated testing techniques (exercising time)

In our experiment, we run the automated tools Sapienz, ACTEve, and MobiGUITAR for 60 minutes each, as discussed in Section 5.5. Choudhary et al[14] explained that, for their experiment, they had to set a time limit for the exercising procedure as none of the tools were able to terminate automatically. Although, we believe that 60 minutes would be a sufficient time frame to demonstrate the performance (in terms of the achieved code coverage) of the testing approaches used in our experiment. From Table 4, we can see that, on average, our proposed *Mobolic* approach achieves higher code coverage within a shorter time compared to the other automated approaches. On average, for the Java code, *Mobolic* achieves 92% within an average time of 22 minutes, whereas Sapienz, ACTEve, and MobiGUITAR achieve 53%, 32%, and 44%, respectively, within an average time of 60 minutes.

Comparing the time taken by every app involved in our experiment, we can see that the maximum time is 58.44 minutes for the "*GM Dice*" app and the minimum time is 3.47 minutes for the "*Guess*" app. Thus, the exact exercising time taken by *Mobolic* for each app is less than 60 minutes, whereas the coverage results are higher in comparison with the other tools. From such observations, we may conclude that, in general, setting a time limit for the GUI testing tools is not a proper solution for the termination of the testing process. We can see that giving a large time frame (60 minutes for each app) does not necessarily guarantee high code coverage results. Based on *Mobolic* performance, we believe that in order to achieve better efficiency (ie, shorter exercising time) and efficacy (higher code coverage), more sophisticated GUI traversal algorithms, input generation mechanisms, and tools underlying implementation are needed.

6 | DISCUSSION

In this section, we discuss the principal differences between *Mobolic*, traditional model-based testing, and symbolic execution. We describe novel aspects brought by *Mobolic* into an automated testing of Android apps. In particular, *Mobolic* integrates the online testing and symbolic execution techniques into one single solution so that it inherits the benefits of both approaches, and thus, it avoids their limitations if they were used separately.

6.1 | How does Mobolic differ from existing model-based testing tools?

Mobolic uses the online testing technique to systematically explore an app GUI by generating relevant events "on the fly" that are immediately executed. Using the online testing technique, it benefits *Mobolic* to potentially execute tests for a long time so that long, intricate, or stressful tests can be performed and significantly reduce the state space (ie, size of the GUI model) to be stored since only a small portion of the GUI model needs to be stored at any point in time. The UI model is automatically extracted from the app and incrementally updated during the testing procedure. In particular, *Mobolic* dynamically builds and updates an *f-GFG* to ensure that the exercising procedure never falls into the infinite loop (since *f-GFG* has no loops) and is always able to finish the testing procedure (since *f-GFG* has finite number of app UIs). *Mobolic* builds the *f-GFG* from the discovered app UIs, which is further used to explore another yet undiscovered app UI.

To efficiently explore the app GUI, *Mobolic* implements A*, an informed search algorithm, which is the best-known form of BFS.[23,42] The A* search algorithm combines BFS for efficiency with the uniform cost search for optimality and completeness. The key idea behind the A* search algorithm is to find the shortest path leading to the target app UI. It is worth noticing that *Mobolic* performs A* search in the *f-GFG*; therefore, *Mobolic* always generates the shortest sequence of UI events leading to the target app UI. In addition, in *f-GFG*, there is always at least one path leading to the target app UI; therefore, *Mobolic* guarantees that the sequence of UI events is always generated.

In sharp contrast, the existing model-based testing tools[14] commonly implement uniformed (standard) *depth*- or *breadth*-first search algorithms[42] as a standalone (ie, either one), or their combination is used for automated GUI exploration. However, in practice, their implementation of *depth*- or *breadth*-first search algorithms does not demonstrate sufficient efficacy to adequately explore the app GUI.[14,15] As a fact, due to the nature of the mobile app GUIs, their models commonly have an infinite number of app UIs and/or loops in the GUI model. Therefore, the existing tools[14,15] with their *depth*- or *breadth*-first search algorithm implementation may fall into the loops or find an infinite number of app UIs preventing the exercising procedure from automatic termination. Thus, in order to adapt the tools in practice and enable them to automatically finish the exercising procedure, the user is required to manually set a termination condition, eg, limit the exercising time, limit the overall number of events to be injected, or limit the depth of GUI exploration. As a result, such limitations may deny the automated tools from performing an adequate GUI exploration. However, in the case of a finite-state app GUI model without loops, using their *depth*- or *breadth*-first search implementation, the tools may effectively explore the app GUIs.

In summary, *Mobolic* is capable of exploring app UIs at deep depth while avoiding falling into an infinite loop or discovering an infinite number of app UIs. Therefore, *Mobolic* addresses **Problem#1** by building *f-GFG* and performing A* search in the *f-GFG* model. In contrast, the existing tools have limited exploration capability since they perform either *depth*- or *breadth*-first search in the innate app UI model that is used "as is" for systematic GUI exploration.

6.2 | How does Mobolic differ from existing symbolic execution tools?

In this section, we omit the discussion on user-predefined inputs since this is a unique property of *Mobolic* and does not exist in the relevant tools we used and, thus, cannot be compared. Instead, we shall focus our discussion on symbolic execution since it is a common technique used by *Mobolic* and the relevant tools we run in our experiment for concrete-input generation. *Mobolic* uses the symbolic execution technique to generate concrete inputs to force the exercising procedure to execute the code along a certain program path. According to Li et al,[43] nearly 99% of statements in real-world programs are user input–*in*dependent, and thus, performing symbolic execution on these statements is not necessary. Therefore, *Mobolic* symbolically executes only UID statements, whereas other user input–*in*dependent statements are executed concretely.

In sharp contrast, the existing symbolic execution tools[8,28,29,44] tend to execute all program paths symbolically, ie, 100% of all statements, which makes the existing tools impractical due to the unacceptable time required to finish the testing

procedure. Therefore, in practice, search of the program paths needs to be *depth*-bounded or *time*-capped.[31,32] It is worth noticing that since *Mobolic* uses the symbolic execution technique, it also copes with symbolic execution problems such as *path explosion*, *path divergence*, and *complex constraints*. In particular, *Mobolic* mitigates the path explosion problem by executing symbolically only UID statements. The UID statements comprise a small portion of the entire app code, and thus, it helps avoid the path explosion problem in most cases. In contrast, the existing tools execute all program paths symbolically. Therefore, path explosion becomes a common problem, and to deal with it, they limit their search of the program paths either by depth or time.

In practice, symbolic execution also suffers from untrustworthy implementation of existing satisfiability modulo theories (SMT) solvers.[45-47] Therefore, path divergence and complex constraints become common problems. To mitigate these problems, we found a solution by using a reliable SMT solver such as Symbolic Math Toolbox in MATLAB.[37] The Symbolic Math Toolbox provides functions for solving and manipulating symbolic math expressions. In fact, the symbolic engine in MATLAB is solid, and such proprietary solution sounds more trustworthy and reliable and could be less buggy than any other existing third-party open-source SMT solvers.[‡‡‡]

In fact, existing open-source SMT solvers only have a limited set of built-in theories because implementing a complete SMT solver may require overwhelming efforts. As a result, the existing open-source SMT solvers do not support all mathematics and symbolic computations. Importantly, by using the symbolic engine in MATLAB, *Mobolic* is capable of solving symbolic constraints in integer and decimal domains, whereas the existing open-source SMT solvers solve symbolic constraints in the integer domain only. Also, *Mobolic* is capable of solving symbolic constraints in the textual domain, whereas the existing open-source SMT solvers commonly do not solve string symbolic constraints.

In summary, *Mobolic* is capable of generating concrete inputs in most cases, except for highly complex symbolic constraints. Therefore, *Mobolic* addresses **Problem#2** by symbolically executing only UID statements and using solid symbolic engine Symbolic Math Toolbox in MATLAB. In contrast, the existing tools have limited capabilities of solving complex symbolic constraints due to the unreliable implementation of the SMT solvers used.

7 | RELATED WORK

In this section, we provide an overview of state-of-the-art automated GUI testing tools that implement random model-based or dynamic symbolic execution testing approaches. To perform a multivariate analysis of the tools, we compare the various aspects of them, as shown in Tables 5, 6, and 7. Each Table holds and compares related properties for the listed tools. For each Table, *Column 1* indicates the name of the tool that implements an approach, and the symbol ⇑ indicates an improvement of a particular property or feature of *Mobolic* over the other listed tools.

Table 5 gives a comparison of the *specifications* of the testing tools. *Column 2* shows the underlying framework on top of which the tool is built. *Column 3* states ease of use of the tool, ie, it states whether the tool works out of the box without any complex configurations or fixes, ie, "Easy"; whether it required some effort, ie, "Medium"; or whether it required a major effort, ie, "Hard," to make it run. This judgment is solely based on our experience, and the evaluation does not have the value of a user study, mainly because most of these tools are just early prototypes. Note that at the moment of writing our paper, *Word2Vec* was not publicly available due to the developer's company policy; therefore, our analysis of the tool is based on its paper content. Thus, for *Word2Vec*, we indicate ease of use as "Unknown" since we were not able to identify ease of use. *Columns 4* and *5* indicate the minimum supported API level of the Android framework requiring the tool to run and the Android device on which the tool can be deployed, respectively. Note that *Word2Vec* is developed for the iOS platform. As such, for this tool, the API level attribute is not applicable; hence, we indicate "N/A." *Columns 6* and *7* state if a discussed tool is self-terminative and what its termination condition is, respectively. If "No," it means that the tool is implemented in such a way that it will infinitely continue app execution unless the user manually specified a termination condition (ie, termination condition is user dependent). For example, the user limits the exercising procedure by time (ie, minutes, hours, days, etc), by the number of events to be injected (ie, UI and/or system events), or by depth of exploration in the app GUI model (ie, limit length of the event sequence to be injected) (*Column 7*). If "Yes," it means that the tool is able to terminate automatically, and thus, a user is not required to manually specify a termination condition (ie, termination condition is user *in*dependent). For example, *Mobolic* innately terminates once it cannot discover any

[‡‡‡] https://en.wikipedia.org/wiki/Satisfiability_modulo_theories\#SMT_solvers

TABLE 5 Comparison of specifications of mobile testing tools

Tool	Underlying Framework(s)	Ease of Use	API Level	Device	Self-Terminative?	Termination Condition
Monkey	UI/Application Exerciser Monkey	Easy	1+	Physical & Emulator	No	Number of injected events
A^3E	Troyd & RERAN	Medium	1+	Physical & Emulator	No	Depth exploration of GUI model
Dynodroid	MonkeyRunner	Easy	10	Emulator	No	Number of injected events
ACTEve	MonkeyRunner	Hard	10	Emulator	No	Execution time
MobiGUITAR	AndroidRipper	Hard	10+	Emulator	No	Execution time
SwiftHand	Chimpchat	Hard	16+	Physical & Emulator	No	Execution time
Mobolic	*UI Automator*	*Easy*	*18+*	*Physical & Emulator*	*Yes⇑*	*Innate exploration of GUI model⇑*
Sapienz	MonkeyRunner	Easy	19	Physical&Emulator	No	Execution time
Word2Vec	iOS Monkey (XCTest)	Unknown	N/A (iOS)	Emulator	No	Execution time

new UIs in the given app GUI model, ie, *Mobolic* has explored as many app UIs in the GUI model as possible, and all the discovered UIs in *f-GFG* are fully exercised.

Self-termination and the termination condition are coupled together in such a way that the termination condition determines if an automated tool can be self-terminative. After our experiments, we would conclude that the *termination condition* is one of the key factors and important properties of the automated tools to increase the chance that the functionality of the app under test is covered as much as possible. An importance of the termination condition is simply explained by the true fact that, beforehand, we are not aware how much, eg, time, events, or GUI depth exploration are needed to cover a particular or all possible app functionalities via automated app GUI exercising. Note that, here, we are discussing about black-box functional UI stress testing and not white-box unit testing where app internals are known to the user (tester) before it runs a test. In comparison with the other listed tools, *Mobolic* implements an improved termination condition that makes *Mobolic* self-terminative.

Table 6 gives a comparison of the common *characteristics* of the testing tool. *Column 2* shows whether the tool requires an Android platform and/or the app under test to be instrumented prior to the app testing. Note that this column indicates only instrumentation that is required as a part of the tool implementation to make it possible to run the app. It does not indicate instrumentation for any other purposes, eg, for code coverage collection, which is not required for app testing. *Column 3* indicates what events are supported by a tool. Note that Monkey, Dynodroid, and ACTEve support a limited number of system events since they are highly structured and vary depending on the Android platform and the app itself. Therefore, there is not a clear way on how to universally generate them on all available Android platform versions and the existing Android apps. *Column 4* shows in which manner a tool generates a user input. A *manual* user input could be provided either during an app testing or could be a predefined set of desired user values that are stored in a tool configuration file for a particular app. An *automated* user input can be either randomly generated or derived based on the heuristics collected from the app, eg, GUI model or app code (eg, source or bytecode), during or before app testing. *Columns 5* and *6* state in which box-mode a tool is able to perform app testing, eg, depending on the source code availability, and which GUI exploration strategy it implements as an underlying testing approach. Generally, black-box requires only an executable source, whereas gray-box may require reversing of the executable source, eg, for instrumentation. The white-box requires the source code to be available. The gray- and white-boxes may limit the tools' practicability due to source code unavailability or the inability to properly reverse an executable source.

It is a fact that mobile app GUIs are highly interactive and commonly require sensible user inputs that are difficult to generate in an automated manner. Thus, we made a step ahead and introduced an improved automated input generation, namely, *Customated*. In comparison with the other listed tools, customated input generation is a fully automated mechanism aiming to generate "*human-like*" user inputs. It consolidates concrete (via performing symbolic execution), user-predefined (via analyzing heuristics [textual attributes] of a relevant editable widget on the foreground screen), and random (numerical or textual) user inputs. As such, customated input generation enables *Mobolic* to more adequately exercise an app functionality in an automated manner.

Table 7 gives a comparison of the *underlying GUI exploration strategies* of the testing tool. In Table 7, *b* is the branching factor, ie, a number of subsequent paths that originate from a given app UI, and *n* is a depth, ie, the number of app UIs in the path leading to an interested app UI. *Column 2* states which GUI explanation strategy is implemented in a tool. *Columns 3* and *4* indicate which GUI search algorithm it involves and what worst-case search time complexity it has, respectively. For random-based GUI exploration strategies, GUI search and worst-case values are not applicable, ie, "N/A."

TABLE 6 Comparison of characteristics of testing tools

Tool	Instrumentation	Events	Input Generation	Box	Approach
Monkey	N/A	UI&System	Automated	Black	Random-based
A^3E	N/A	UI	Manual	Black&Gray	Model-based
Dynodroid	Platform	UI&System	Automated&Manual	Black	Random-based
ACTEve	Platform&App	UI&System	Automated	White	Dynamic symbolic
MobiGUITAR	N/A	UI	Manual	Gray	Model-based
SwiftHand	App	UI	Automated&Manual	Gray	Model-based
Mobolic	*N/A*	*UI&System*	*Customated⇑*	*Black*	*Model-based*
Sapienz	App	UI	Automated	Black&Gray&White	Search-based
Word2Vec	N/A	UI	Automated	Black	Model-based

TABLE 7 Comparison of graphical user interface (GUI) exploration strategies of testing tools

Tool	GUI Exploration	GUI Search	Worst-Case
Monkey	Random	N/A	N/A
A^3E	Systematic	Depth-first	$\mathcal{O}(b^n)$
Dynodroid	Improved Random	N/A	N/A
ACTEve	Improved Systematic	N/A	N/A
MobiGUITAR	Systematic	Depth&Breadth-first	$\mathcal{O}(b^n)$
SwiftHand	Systematic	Depth-first	$\mathcal{O}(b^n)$
Mobolic	*Improved Systematic*	$A^*\Uparrow$	$\mathcal{O}(n)\Uparrow$
Sapienz	Improved Random	N/A	N/A
Word2Vec	Systematic	Depth-first	$\mathcal{O}(b^n)$

In comparison with the listed tools, *Mobolic* significantly reduces search time in *f-GFG* model by involving an informed search algorithm A*.[23,42] In *Mobolic*, we implemented A* search in such a way that its time complexity is $\mathcal{O}(n)$, which is linear in the number of app UIs (n) on the path leading to the lastly discovered app UI. Thus, A* enables *Mobolic* to build the shortest UI-path to quickly reach the lastly discovered app UI. As such, *Mobolic* is able to continue the exercising process from the lastly discovered app UI and exercise all its yet unexercised widgets, and to discover new, yet undiscovered, app UIs (if any). Such the improvement makes *Mobolic* efficient and effective in terms of the exercising time and achieved code coverage.

Apart from our comparison, there is another research work[15] which performs a statistical analysis, and comparison of various attributes and properties of multiple automated testing techniques for Android mobile apps. The work offers a comprehensive in-depth comparison list of the testing tools existing in the literature including those which are used in our paper. It establishes a general framework which abstracts all the common characteristics of online testing techniques proposed in the literature. In particular, it shows how the framework can be used to design experiments aimed at performing objective comparisons among different online testing approaches, and how it helps to identify an influence of different tools parameters on the performance of the testing techniques.

7.1 | Random testing

Random testing[5,48,49] is the most simple approach to exercising mobile apps. Random testing typically starts by creating a simplified model (or without model) of the app under test. The built model can be used to generate the random inputs and/or sequences of actions. The random testing can be *guided* or *unguided*. The unguided (undirected) random testing does not have heuristics to guide its search. The guided (directed) random testing extracts heuristics from the app under test to guide its search and possibly input generation, eg, *feedback-directed*[50] or *adaptive*[51,52] random testing. In practice, there are several testing tools such as Randoop,[53] Artemis,[54] Dynodroid,[9] EvoDroid,[27] and DART[44] that implement *feedback-directed* automated random testing with an event-prioritizing mechanism for mobile app testing.

Monkey[§§§ 55-57] does not require any knowledge of the app GUI model. It simply generates a sequence of pseudo-random UI events such as clicks, touches, or gestures as well as a limited number of system events. To run Monkey, the user requires to set a number of events to be injected. It terminates once all events have been injected. The Monkey does not consider any app state or UI transitions and simply performs random actions at random positions on the UI. Therefore, adequacy of Monkey could be affected by the density and/or the physical size of the UI widgets on the app UIs. Also, Monkey is not able to generate relevant to the current app state input that is commonly required for the mobile apps due to the highly interactive nature of GUIs. Thus, generally, it is a challenge for random tests to adequately exercise app functionality without intelligent input generation.

Dynodroid[9] implements an *improved* random-based GUI exploration strategy compared to Monkey. It can either select the events that have been least frequently selected (*Frequency* strategy) or keep into account the context (*BiasedRandom* strategy), ie, events that are relevant in more contexts will be selected more frequently. Compared to Monkey, Dynodroid generates UI and a limited set of system events that are relevant to the current app state. In addition, Dynodroid allows

[§§§] http://developer.android.com/tools/help/monkey.html

the users to manually provide inputs (eg, login/password) when concrete ones are required. Same as Monkey, Dynodroid does not terminate automatically; therefore, it requires the user to manually set a certain number of events to be injected as a termination condition.

7.2 | Model-based testing

Model-based testing approaches[5,6,58-62] use a UI model that is derived from the app GUI 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. For example, A^{2}T^2 [63] implements a default random testing mechanism guided by the inferred app UI model. In particular, it relies on a GUI crawler to mimic the actual user events on the app GUI and automatically infer the GUI model. The GUI crawler builds the GUI model based on the extracted *Event-Flow Graph* abstraction of the fireable events on the app UIs. That is, the crawling procedure operates using the fundamental entity "*Event*" to explore the app GUI. It uses the extracted events with their preconditions (event sequences), which are to be executed from the root to the leaves of the GUI tree, ie, from the first app UI showing upon app launch to the subsequent app UIs. Hence, due to such design of the GUI crawler, it does not concern itself with the 2 important innate issues existing in the automated GUI exploration. Firstly, it does not consider the problem of multiple app restarts.[11] Secondly, it does not consider re-exercising of the previously discovered app UIs multiple times (ie, exercise the same app UIs more than only once depending on the GUI flow). In particular, in order to execute every extracted event sequence, the GUI crawler requires multiple app restarts that could make the exercising process notably lengthy. In addition, the GUI crawler does not consider re-exercising the same app interfaces if they were previously discovered. Thus, the GUI crawling algorithm may overlook certain app interfaces, especially if they are located at deep levels in the GUI model.

MobiGUITAR[7] is an automated GUI-driven testing framework for Android apps. The MobiGUITAR is based on the 3-stage principal such as "*observation*," "*extraction*," and "*abstraction*" of the runtime state of UI widgets. The "*abstraction*" is a scalable state-machine model that, together with test coverage criteria, helps automatically generate functional test cases. MobiGUITAR implements a *breadth*-first and *depth*-first search algorithms for traversing the app UI model. It restarts the exercising procedure from the starting app state when it cannot find any new ones. In practice, a restart can be time consuming[11] for most of real-world Android apps. MobiGUITAR requires major user efforts to manually configure multiple tool parameters.[14]

A^{3}E[10] aims to systematically discover new app states with high coverage via 2 techniques, namely, *Depth*-first and *Targeted* explorations, to improve method and activity coverage, respectively. The *Depth*-first exploration is a dynamic approach that is based on the automated exploration of GUI activities and their widgets in a depth-first manner. It builds the abstract GUI model that represents each activity as a single app state, without considering different states of the widgets on the activity. Thus, this abstraction may lead to missing certain app functionalities that would be easy to exercise if a more concrete model was used. The *Targeted* exploration is a directed approach that allows to explore activities more efficiently by generating relevant intents. It uses static bytecode analysis to build the Static Activity Transition Graph (SATG) of the app, and then, it systematically explores the SATG while the app runs on the mobile device (real or emulator). By solely generating relevant intents, only a limited number of activities can be explored. Certain activities (eg, login activity) could be constrained by the specific user input so that all subsequent activities will not be discovered unless the relevant user input is provided.

SwiftHand[11] aims to achieve code coverage quickly by learning and exploring an abstraction of the app GUI model. It learns the GUI model by exploiting execution traces generated during the testing process and applying the state-merging deterministic finite automaton induction machine learning algorithm. *SwiftHand* uses the learned GUI model to choose inputs that lead to yet undiscovered app states. Also, it builds the GUI model to minimize the number of app restarts by searching for paths to reach new app states using only UI inputs.

Sapienz[13] is an approach to automated Android testing that uses multiobjective search-based testing. It automatically explores and optimizes event sequences and minimizes their length while simultaneously maximizing code coverage and fault detection. Sapienz extends model-based testing by employing search-based exploration and exploits automated random user input seeding and multilevel app instrumentation. Sapienz minimizes the event sequence length and maximizes other objectives that can be combined in a Pareto-optimal multiobjective search-based approach. By using Pareto optimality, it does not sacrifice longer event sequences, when they are the only ones that find faults, nor where they are necessary to achieve higher code coverage. Through its use of Pareto optimality, Sapienz progressively replaces such longer sequences with shorter ones when their performance is equivalent.

Word2Vec[62] is a novel deep learning–based approach that addresses a problem of automatic generation of the most *relevant* text inputs in a use case context and that is applicable on a large scale as well. The *relevance* is specific to the natural-language semantics that only humans can understand. As such, the approach focuses on natural-language processing using a recurrent neural network (RNN) and *Word2Vec* models, where traditional automated input generation approaches such as symbolic execution are not applicable. For automated systematic GUI exploration, the developed approach uses the XCTest, an iOS Monkey engine, which, in turn, adopts the *Depth*-first search strategy. For automated input generation, the developed approach leverages a combination of the RNN and *Word2Vec* models to find meaningful text inputs that are related to the current app context.

The approach demonstrates its effectiveness. It measures the effectiveness by computing the screen coverage, ie, it counts the number of different (unique) UI screens that have been explored within a fixed time limit. The results show that, in terms of the achieved screen coverage, using the automated input generation, the RNN model outperforms the random one by 46%, whereas the combination of the RNN and *Word2Vec* models outperforms it by 60%. The difference between the RNN model and the combination of the RNN and *Word2Vec* models highlights the effectiveness of using the *Word2Vec*.

7.3 | Dynamic symbolic execution

Symbolic execution[64-66] is another software testing technique. Its objective is to systematically discover as many execution paths in a program as possible. The main idea of symbolic execution is to replace concrete values with symbolic expressions that can assume any possible value. Therefore, symbolic execution theoretically may explore all possible paths through the program and generate test cases to achieve high structural coverage. In practice, symbolic execution has severe problems such as *constraint complexity*, *path divergence*, and *path explosion*.[30] Furthermore, symbolic execution generally requires the user to instrument the app or execution environment before testing.

ACTEve[8] performs automated testing for Android applications using concolic execution. ACTEve explores the apps starting from their entry point, but does not aim for particular targets, and employs concolic execution at the level of the entire app rather than on individual event handlers. ACTEve symbolically tracks events from the point in the Android framework where they are generated up to the point where they are handled in the app. Also, ACTEve uses concolic execution for reasoning about low-level properties of events (eg, coordinates of UI widgets), which it can treat more abstractly by using the UI models.

8 | CONCLUSION

In this work, we have proposed a novel automated GUI testing technique, namely, *Mobolic*, to achieve high code coverage for Android app testing by combining the online testing technique and customated input generation. We implemented *Mobolic* for Android apps, considering their specific nontrivial structure and the highly interactive nature of GUIs. We evaluated the performance of *Mobolic* on 10 real-world open-source Android apps and compared it with prevalent approaches, ie, manual testing (Human) and automated ones (Sapienz, ACTEve, and MobiGUITAR). Our experimental results show that, on average, *Mobolic* is capable of achieving high code coverage of 92% within a time limit of 22 minutes on average, whereas Human, Sapienz, ACTEve, and MobiGUITAR achieved 78%, 54%, 32%, and 44%, respectively, within a time limit of 60 minutes.

In our future work, we are planning to further improve the *smartCE* customated input generation. In particular, we will improve the accuracy and reliability of concrete-input generation. Thus, we further improve its ability to solve more complex variations of symbolic string constraints, ie, constraints whose solutions lie in a textual domain. In addition, we will expand a set of supported commonly used input types that are needed for the user-predefined inputs. Also, we will expand a limited number of currently supported system events to enable *Mobolic* to exercise service apps as well, ie, apps that do not have a GUI. As a result, we will improve our tool reliability and practicability, widen a range of mobile apps that could be exercised, and further increase code coverage approaching 100% of covered Java app code.

ORCID

Yauhen Leanidavich Arnatovich http://orcid.org/0000-0001-8266-9151

REFERENCES

1. Gianazza A, Maggi F, Fattori A, Cavallaro L, Zanero S. *Puppetdroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications*. 2014. arXiv preprint arXiv:1402.4826.
2. Hu C, Neamtiu I. Automating GUI testing for Android applications. Paper presented at: Proceedings of the 6th International Workshop on Automation of Software Test; 2011; Honolulu, HI.
3. Nguyen BN, Robbins B, Banerjee I, Memon A. GUITAR: an innovative tool for automated testing of GUI-driven software. *Autom Softw Eng*. 2014;21(1):65-105.
4. Yuan X, Memon AM. Using GUI run-time state as feedback to generate test cases. Paper presented at: Proceedings of the 29th International Conference on Software Engineering (ICSE'07); 2007; Washington, DC.
5. Yang W, Chen Z, Gao Z, Zou Y, Xu X. GUI testing assisted by human knowledge: random vs. functional. *J Syst Softw*. 2014;89:76-86.
6. Gutiérrez J, Escalona M, Mejías M. A model-driven approach for functional test case generation. *J Syst Softw*. 2015;109:214-228.
7. Amalfitano D, Fasolino AR, Tramontana P, Ta BD, Memon AM. MobiGUITAR: automated model-based testing of mobile apps. *IEEE Softw*. 2015;32(5):53-59.
8. Anand S, Naik M, Harrold MJ, Yang H. Automated concolic testing of smartphone apps. Paper presented at: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Vol. 59; 2012; Cary, NC.
9. Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for Android apps. Paper presented at: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering; 2013; Saint Petersburg, Russia.
10. Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of Android apps. Paper presented at: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications; 2013; Indianapolis, IN.
11. Choi W, Necula G, Sen K. Guided GUI testing of Android apps with minimal restart and approximate learning. Paper presented at: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications; 2013; Indianapolis, IN.
12. Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM. Using GUI ripping for automated testing of Android applications. Paper presented at: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering; 2012; Essen, Germany.
13. Mao K, Harman M, Jia Y. Sapienz: Multi-objective automated testing for Android applications. Paper presented at: Proceedings of the 25th International Symposium on Software Testing and Analysis; 2016; Saarbrücken, Germany.
14. Choudhary SR, Gorla A, Orso A. Automated test input generation for Android: Are we there yet? Paper presented at: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2015; Lincoln, NE.
15. Amalfitano D, Amatucci N, Memon AM, Tramontana P, Fasolino AR. A general framework for comparing automatic testing techniques of Android mobile apps. *J Syst Softw*. 2017;125:322-343.
16. Utting M, Pretschner A, Legeard B. A taxonomy of model-based testing approaches. *Softw Test Verification Reliab*. 2012;22(5):297-312.
17. Larsen KG, Mikucionis M, Nielsen B. *Online Testing of Real-Time Systems Using Uppaal*. Berlin, Germany: Springer Berlin Heidelberg; 2005:79-94.
18. Linehan MH. Semantics in model-driven business design. Paper presented at: Proceedings of the 2nd International Semantic Web Policy Workshop; 2006; Yorktown Heights, NY.
19. Lee J. Model-driven business transformation and the semantic web. *Commun ACM*. 2005;48(12):75-77.
20. Gamboa MA, Syriani E. Automating activities in MDE tools. Paper presented at: 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD); 2016; Rome, Italy.
21. Klein J, Levinson H, Marchetti J. Model-driven engineering: automatic code generation and beyond [Technical report]. Pittsburgh, PA: Software Engineering Institute at Carnegie Mellon University; 2015.
22. Arnatovich YL, Ngo MN, Kuan THB, Soh C. Achieving high code coverage in Android UI testing via automated widget exercising. Paper presented at: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC); 2016; Hamilton, New Zealand.
23. Potdar GP, Thool R. Comparison of various heuristic search techniques for finding shortest path. *Int J Artif Intell Appl*. 2014;5(4):63.
24. Jensen CS, Prasad MR, Møller A. Automated testing with targeted event sequence generation. Paper presented at: Proceedings of the 2013 International Symposium on Software Testing and Analysis; 2013; Lugano, Switzerland.
25. Nguyen CD, Marchetto A, Tonella P. Combining model-based and combinatorial testing for effective test case generation. Paper presented at: Proceedings of the Combining International Symposium on Software Testing and Analysis; 2012; Minneapolis, MN.
26. Yang W, Prasad MR, Xie T. Grey-box approach for automated GUI-model generation of mobile applications. *Fundamental Approaches to Software Engineering*. Rome, Italy: Springer; 2013:250-265.
27. Mahmood R, Mirzaei N, Malek S. EvoDroid: Segmented evolutionary testing of Android apps. Paper presented at: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2014; Hong Kong, China.
28. Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: automatically generating inputs of death. *ACM Trans Inf Syst Secur*. 2008;12(2):1-38.
29. Inkumsah K, Xie T. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. Paper presented at: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE'07); 2007; New York, NY.
30. Anand S. Techniques to Facilitate Symbolic Execution of Real-World Programs [PhD thesis]. Atlanta, GA; Georgia Institute of Technology: 2012.

31. Siddiqui JH, Khurshid S. Parsym: Parallel symbolic execution. Paper presented at: 2010 2nd International Conference on Software Technology and Engineering; 2010; San Juan, Puerto Rico.
32. Staats M, Pasareanu C. Parallel symbolic execution for structural test generation. Paper presented at: Proceedings of the 19th International Symposium on Software Testing and Analysis; 2010; Trento, Italy.
33. Scholz B, Zhang C, Cifuentes C. User-input dependence analysis via graph reachability [Technical report]. Mountain View, CA; 2008.
34. Elish KO, Yao D, Ryder BG. User-centric dependence analysis for identifying malicious mobile apps. Paper presented at: Workshop on Mobile Security Technologies; 2012; San Francisco, CA.
35. Wei F, Roy S, Ou X, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. Paper presented at: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security; 2014; Scottsdale, AZ.
36. Hao S, Li D, Halfond WG, Govindan R. SIF: A selective instrumentation framework for mobile applications. Paper presented at: Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'13); 2013; Taipei, Taiwan.
37. Symbolic Math Toolbox. https://www.mathworks.com/help/symbolic. Accessed January; 2017.
38. Runeson P, Host M, Rainer A, Regnell B. *Case Study Research in Software Engineering: Guidelines and Examples*. Hoboken, NJ: John Wiley & Sons; 2012.
39. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Berlin, Germany: Springer Science & Business Media; 2012.
40. Feldt R, Magazinius A. Validity threats in empirical software engineering research-an initial survey. SEKE; 2010.
41. Wright HK, Kim M, Perry DE. Validity concerns in software engineering research. Paper presented at: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10); 2010; Santa Fe, NM.
42. Poole DL, Mackworth AK. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge, UK: Cambridge University Press; 2010.
43. Li G, Lu K, Zhang Y, Lu X, Zhang W. Mixing concrete and symbolic execution to improve the performance of dynamic test generation. Paper presented at: 2009 3rd International Conference on New Technologies, Mobility and Security; 2009; Cairo, Egypt.
44. Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. Paper presented at: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05); 2005; Chicago, IL.
45. De Moura L, Bjørner N. Z3: An efficient SMT solver. Paper presented at: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems; 2008; Hungary, Budapest.
46. De Moura L, Bjørner N. Satisfiability modulo theories: An appetizer. Paper presented at: 12th Brazilian Symposium on Formal Methods; 2009; Gramado, Brazil.
47. Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. Paper presented at: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems; 2011; Newport Beach, CA.
48. Hamlet R. Random testing. Encyclopedia of software Engineering; 1994.
49. Zhang J, Cheung SC. Automated test case generation for the stress testing of multimedia systems. *Softw Pract Exp*. 2002;32(15):1411-1435.
50. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. Paper presented at: Proceedings of the 29th International Conference on Software Engineering; 2007; Minneapolis, MN.
51. Chen TY, Leung H, Mak I. Adaptive Random Testing. *Advances in computer science-ASIAN. Higher-Level Decision Making*. Chiang Mai, Thailand: Springer; 2004:320-329.
52. Liu H, Xie X, Yang J, Lu Y, Chen TY. Adaptive random testing through test profiles. *Softw Pract Exp*. 2011;41(10):1131-1154.
53. Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for Java. Paper presented at: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion; 2007; Montreal, Canada.
54. Artzi S, Dolby J, Jensen SH, Moller A, Tip F. A framework for automated testing of JavaScript web applications. Paper presented at: 2011 33rd International Conference on Software Engineering (ICSE); 2011; Honolulu, HI.
55. Nyman N. Using monkey test tools. *Software Testing & Quality Engineering Magazine*; 2000.
56. Hofer B, Peischl B, Wotawa F. GUI savvy end-to-end testing with smart monkeys. Paper presented at: 2009 ICSE Workshop on Automation of Software Test; 2009; Vancouver, Canada.
57. Brummayer R, Lonsing F, Biere A. *Automated Testing and Debugging of SAT and QBF Solvers*. Berlin, Germany: Springer Berlin Heidelberg; 2010:44-57.
58. Mehlitz P, Tkachuk O, Ujma M. JPF-AWT: Model checking GUI applications. Paper presented at: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering; 2011; Lawrence, KS.
59. White L, Almezen H. Generating test cases for GUI responsibilities using complete interaction sequences. Paper presented at: Proceedings of the 11th International Symposium on Software Reliability Engineering; 2000; San Jose, CA.
60. Marback A, Do H, He K, Kondamarri S, Xu D. A threat model-based approach to security testing. *Softw Pract Exp*. 2013;43(2):241-258.
61. Walton GH, Poore JH. Generating transition probabilities to support model-based software testing. *Softw Pract Exp*. 2000;30(10):1095-1106.
62. Liu P, Zhang X, Pistoia M, Zheng Y, Marques M, Zeng L. Automatic text input generation for mobile testing. Paper presented at: Proceedings of the 39th International Conference on Software Engineering (ICSE'17); 2017; Buenos Aires, Argentina.
63. Amalfitano D, Fasolino AR, Tramontana P. GUI crawling-based technique for Android mobile application testing. Paper presented at: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops; 2011; Berlin, Germany.

64. Cadar C, Godefroid P, Khurshid S, et al. Symbolic execution for software testing in practice: Preliminary assessment. Paper presented at: Proceedings of the 33rd International Conference on Software Engineering; 2011; Honolulu, HI.

65. Mirzaei N, Malek S, Păsăreanu CS, Esfahani N, Mahmood R. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*. 2012;37(6):1-5.

66. Godboley S, Mohapatra DP, Das A, Mall R. An improved distributed concolic testing approach. *Softw Pract Exp*. 2017;47(2):311-342.

How to cite this article: Arnatovich YL, Wang L, Ngo NM, Soh C. Mobolic: An automated approach to exercising mobile application GUIs using symbiosis of online testing technique and customated input generation. *Softw Pract Exper*. 2018;1–36. https://doi.org/10.1002/spe.2564