# User Guided Automation for Testing Mobile Apps

Xiujiang Li*†, Yanyan Jiang*†, Yepang Liu‡, Chang Xu*†[1], Xiaoxing Ma*†[1], Jian Lu*†
*State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
†Dept. of Computer Science and Technology, Nanjing University, Nanjing, China
‡Dept. of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China
njucslxj@gmail.com, jiangyy@outlook.com, andrewust@cse.ust.hk, {changxu,xxm,lj}@nju.edu.cn

***Abstract*—Mobile devices are gradually taking over traditional computers' dominance in human lives. With the ever-increasing shipment of mobile apps running on these devices, their quality issues become a severe challenge. Although automated testing techniques are being widely studied, they mostly fall short of handling mobile apps' complex interactions, e.g., a finger swipe or device shaking gesture, leading to inadequate testing. In this paper, we present a novel User Guided Automation (UGA) technique to address testing challenges incurred by such complex interactions. UGA exploits user insights to complement automated testing techniques by recording user-guided app executions, replaying apps to certain stop points, and systematically exploring state space from these stop points. We implemented our work as a prototype UGA tool on Android platform and evaluated it on seven real-world Android apps. Evaluation results show that UGA achieved 1.59–21.78× improvement in terms of method coverage over state-of-the-art automated techniques in testing mobile apps.**

***Index Terms*—Android apps; Testing; Semi-automated; User trace.**

## I. Introduction

Mobile devices such as smartphones and tablets have become the de facto computers in our daily lives [1]. One major reason behind their market success is that apps running on these devices can assist their users in many daily activities (e.g., work and entertainment) and provide rich interactive experiences [2]. Such benefits, on the other hand, have also challenged traditional testing practice because testing apps with complex user interactions is always labor-intensive [3]. Moreover, competitive app markets force developers to quickly release their apps. Hence, developers rarely have adequate time for thorough app testing [4]. Due to these reasons, automated testing or testing assistance techniques are highly desirable.

In recent years, researchers have proposed various automated testing techniques [5]–[11]. However, such automation still remains at a methodology level. Although it is theoretically sound to treat complex interactions as transitions in a GUI model, whether such abstraction can be automated (e.g., whether a sensor-involving interaction can be realized in testing easily) is still an open question. Therefore, when existing tools are applied to real-world apps, they often require complex manual interactions (e.g., conducting a finger swipe or device shaking gesture, or typing in user credentials) to reach certain GUI entries. This significantly reduces the attractiveness of these existing test automation techniques.

[1]Corresponding authors.

Complex interactions thus hinder test automation, but performing them is instead not difficult for human beings. To alleviate the above limitation of existing automated testing techniques, we in this paper present a novel user guided test automation technique (named UGA), which integrates user interactions into automated testing. UGA takes advantage of automated testing to systematically traverse different app states. At the same time, it also exploits the insight that users can easily understand an app's intentions and interact with an app effectively to cover code regions or app states that can hardly be reached by pure automated techniques.

Our UGA technique works in two phases. First, UGA records an app's execution trace when its user executes this app, and we name it a user trace. This record phase saves log that can be used later for reproducing this user trace as well as extending it to other potential traces. Second, UGA replays a partial trace of a previously logged execution to a stop point (i.e., an app state where its user spends much time on inspecting GUI, or can switch to another new GUI). Starting from this stop point, UGA performs automated testing (e.g., random or depth-first exploration) to systematically explore the app states around this stop point. In this way, by leveraging user traces to reach certain stop points, which cannot easily be reached without complex user interactions, and by performing systematic testing from those stop points, one can have a higher chance of visiting more states than simply applying a pure automated testing technique.

To validate our idea, we implemented our UGA in Java for the popular Android platform. To support UGA, we also integrated it with RERAN [12] to collect user traces, and with standard random testing RND [8], [9], [13] and model-based depth-first systematic testing DFS [6], [11] such that UGA can leverage these two strategies for testing in its second phase. We evaluated our tool (also named UGA for convenience) experimentally on seven real-world popular Android apps. The experimental results consistently showed that UGA can significantly improve method coverage in testing (1.62–14.51× improvement on RND and 1.59–21.78× improvement on DFS). These results suggest that guiding automated testing by exploiting easily-obtained user traces is a promising direction towards cost-effective testing for mobile apps.

In summary, our contributions in this paper are two-fold. First, we propose to exploit user traces to complement automated testing techniques for testing mobile apps that require complex interactions. Second, we implemented our technique

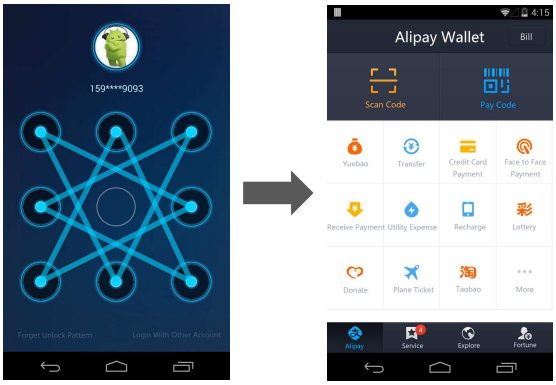



Figure 1: First two screens (s1 and s2) in Alipay Wallet

and evaluated its effectiveness through real-world apps. The results confirm that the effectiveness of existing automated testing techniques can be greatly enhanced by leveraging a small amount of user traces, which can be collected with negligible cost.

The rest of this paper is organized as follows. Section II overviews our UGA idea with a concrete example. Section III elaborates on our UGA technique and Section IV explains its implementation. Section V evaluates our UGA technique experimentally with real-world apps. Section VI discusses related work and explores our future work, and finally, Section VII concludes this paper.

## II. Overview

In this section, we present a motivating example. We use this example to discuss limitations of existing automated testing techniques for mobile apps. We then explain how our UGA technique can address such limitations in a user-guided way.

### A. A Motivating Example

We use Alipay Wallet [14], a popular online payment app in China, as our motivating example. Figure 1 shows the first two screens after Alipay Wallet starts. We denote them as s1, s2, respectively. Since the app involves financial transactions, for safety concerns, it requires user authentication by drawing an unlock pattern on the screen (s1). Only correct patterns can lead to the app's main screen (s2).

To test such an app, existing automated testing techniques would require a GUI model that describes how users can interact with this app at each of its possible states. Such a model may be derived by analyzing an app's GUI layout at runtime (e.g., by tools Robotium [15] and Hierarchy Viewer [16]), or by analyzing the app's source code [11]. Based on thus obtained GUI models, existing techniques can then explore the app's state space by feeding necessary input actions. Although this sounds straightforward and many simple input actions (e.g., button click) can indeed be simulated, such testing techniques do not work effectively for many real-world apps. The reason is that many apps require complex interactions, e.g., typing in a valid account/password combination or performing a particular drawing gesture as in Alipay Wallet. As a result, the underlying code for handling such complex interactions becomes a dead zone for such automated techniques. In fact, we did observe that none of existing automated testing techniques for mobile apps can reach screen s2 in our motivating example. While it is possible for professional developers to write specific test drivers to bypass such interactions, this treatment contradicts our expectation of test automation.

### B. Our User-guided Technique

Our UGA technique exploits a simple yet effective observation that developers often run their apps to see whether the apps' major functionalities work or not before applying sophisticated testing tools. These initial runs typically contain critical information about complex interactions, and can well cover major functionalities of an app. Based on this observation, our UGA first records user traces during such initial runs, and use them to guide its automated testing later. Specifically, in its automated testing phase, UGA replays its recorded user traces to certain stop points where users (e.g., testers) stop for choosing next actions. When reaching any of such stop points, UGA's replay engine would hand over the execution control to its automated testing engine for a systematic state exploration.

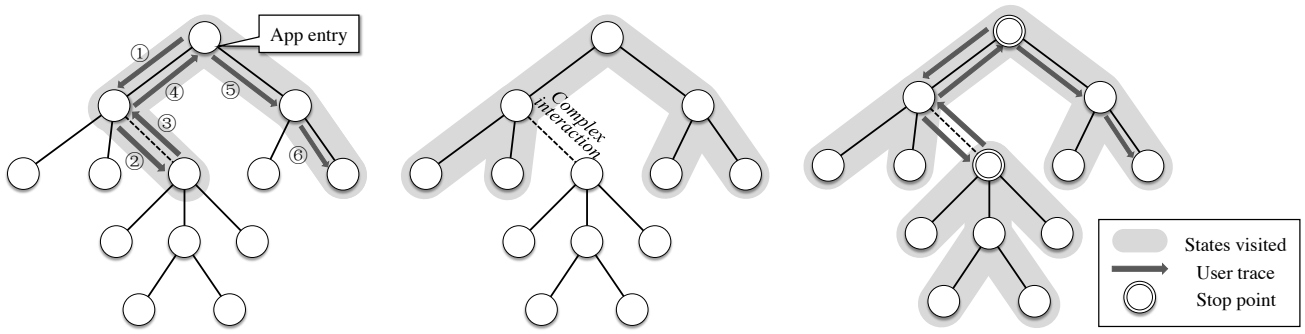
To ease understanding, let us revisit our motivating example. We invite several users to play with Alipay Wallet and our UGA can record user traces accordingly. The users would draw correct unlock patterns and then start to explore functionalities provided by Alipay Wallet. Based on these user traces, UGA can effectively test this app in an automated way. Specifically, it would first select s2 as a stop point (stop point selection criteria are discussed later in Section III) and replay recorded user traces to s2. Then it switches execution to its internal testing engine for systematic exploration from this stop point. On the other hand, without such automated record and replay support, existing testing techniques would be stuck at s1 and can hardly break through it and test later functionalities in this app.

## III. The UGA Approach

In this section, we elaborate on our UGA technique for effective testing of mobile apps.

### A. High-level Idea

Figure 2 illustrates the high-level idea of our UGA technique. It is well-known that a real-world app's state space is typically huge or even infinite. In practice, due to resource restrictions, it is often infeasible for human tester to traverse an app's entire state space, as illustrated in Figure 2(a). An automated testing tool can certainly explore part of an app's state space, but may also lose its direction when its exploration falls into part of the state space whose structure is unclear. Besides, it cannot reach those states that have to be triggered by complex interactions, resulting in low test coverage, as shown in Figure 2(b). Nevertheless, things can get changed if we have user traces at hand. Users can quickly understand how to interact with an app with very little information (e.g., a hint on screen saying "please unlock"), and perform corresponding




(a) user trace only (b) automated testing only (c) user guided automation (UGA)

Figure 2: Illustration for the UGA technique

interactions to reach those states that cannot easily be reached by existing automated testing tools. Moreover, users usually focus on major functionalities when they use an app. Such behavior can help expose the main structure of an app's state space. Guided by such a structure, our UGA can then choose certain stop points to conduct automated testing, thus exploring the app's state space in a systematic way, which can further investigate those states users do not typically pay attention to, i.e., those states associated with the app's minor functionalities. Such an example is demonstrated in Figure 2(c).

Specifically, our UGA works in three steps: (1) user trace recording, (2) stop-point identification, and (3) user-guided automated testing. We in the following elaborate on these three steps.

| | Time | Device | Parameters (type, code and value) | |
|---|---|---|---|---|
| 1. | [500.821153] | /dev/input/event2: | 0003 0039 00000398 | Press |
| 2. | [500.821184] | /dev/input/event2: | 0003 0035 000003eb | |
| 3. | [500.821184] | /dev/input/event2: | 0003 0036 00000478 | |
| 4. | [500.821184] | /dev/input/event2: | 0003 003a 00000032 | |
| 5. | [500.821214] | /dev/input/event2: | 0000 0000 00000000 | |
| 6. | [500.831011] | /dev/input/event2: | 0003 0039 ffffffff | Release |
| 7. | [500.831042] | /dev/input/event2: | 0000 0000 00000000 | |
| 8. | [502.778547] | /dev/input/event2: | 0003 0039 00000399 | Press |
| 9. | [502.778547] | /dev/input/event2: | 0003 0035 00000535 | |
| 10. | [502.778577] | /dev/input/event2: | 0003 0036 00000489 | |
| 11. | [502.778577] | /dev/input/event2: | 0003 003a 00000031 | |
| 12. | [502.778577] | /dev/input/event2: | 0000 0000 00000000 | |
| 13. | [502.930844] | /dev/input/event2: | 0003 0039 ffffffff | Release |
| 14. | [502.930874] | /dev/input/event2: | 0000 0000 00000000 | |
| 15. | [504.000002] | … | | |

Figure 3: Illustration of an user trace

### B. User Trace Recording

UGA records detailed logs of a user's execution (defined as a *user trace*) for later replay. For recording purposes, we adopt the technique described in RERAN [12]. The technique records all input events during the execution of an app, covering how a user uses an app, both intentionally (e.g., screen touching and tapping events) and unintentionally (e.g., sensor reading events). This enables our UGA to faithfully reproduce all complex interactions by an app's user (e.g., press-release, press-hold, swipe, zoom-pinch and shake of a device).

We define each log entry as an event $e = (t, d, p)$, meaning that at time $t$ an input device $d$ creates this event $e$ with parameters $p$. By this definition, we can use $e.t$, $e.d$ and $e.p$ to represent the event $e$'s occurring time, its input device that creates this event and the event's parameters, respectively. UGA logs a sequence of chronologically ordered events $T = \{e_1, e_2, \ldots, e_n\}$ (i.e., $e_1.t \leq e_2.t \leq \ldots \leq e_n.t$) during its record phase, and this sequence is a user trace for an app. Later in its replay phase, UGA feeds the app with events recorded in a user trace according to their timings. Figure 3 gives one example of such user trace, which contains 14 low-level events, representing two press-release interactions.

Each event $e$ in Figure 3 has five fields in total. The first field indicates $e.t$, the duration since the last system restart (i.e., the occurring time of this event). The second field specifies $e.d$, the input device that has created this event. The remaining three fields describes $e.p$, which is a three-tuple. All these events are from the touch screen (`/dev/input/event2`). The three components of $e.p$ represent the event type, the data type, and the actual value. For example, $e.p$ = (0003,0035,000003eb) indicates a touching event's absolute (0003) X coordinate (0035) is 000003eb.

Note that in this recording process, users only need to perform a simple walk-through use of an app. This does not require professional skills, and users are unaware of this recording process. Typically, the recording can be done in a few minutes. This is the only user participation part of our UGA technique. After that, UGA is fully automated.

### C. Stop-point Identification

Now we move to stop-point identification. Suppose that UGA has recorded a user trace $T = \{e_1, e_2, \ldots, e_n\}$. In theory, when later UGA replays this user trace, it can stop for systematic state exploration (i.e., automated testing) after replaying to any one of these events $e_k$ ($k \leq n$). However, in practice, a user trace can contain thousands of logged events. Stopping at each of thus logged events is not only impractical, but also unnecessary for an effective testing. Our

**Algorithm 1:** The RND Algorithm

```
1  function RND(v, ℓ)
2  begin
3      for i ∈ {1, 2, ..., ℓ} do
4          A ← GetAllEnabledActions(v);
5          if A ≠ ∅ then
6              a ← randomChoose(A);
7              execute a, leading to activity v′;
8              v ← v′;
9          else
10             break;
```

**Algorithm 2:** The DFS Algorithm

```
   Data: global variable V with initial value ∅
1  function DFS(v)
2  begin
3      V ← V ∪ {v};
4      for each action a ∈ GetAllEnabledActions(v) do
5          execute a, leading to activity v′;
6          if v′ ∉ V then
7              DFS(v′);
8          if v ≠ v′ then
9              back to activity v;
```

UGA proposes to start automated testing only after replaying to certain logged events. Such events are named *stop points*. Specifically, UGA identifies stop points by the following two heuristic rules:

1) An event $e_k \in T$ is selected as a stop point if after executing event $e_k$, a new screen pops up (i.e., a new activity is created in Android) or a new GUI container is dynamically created. The intuition behind this rule is that automated testing tools are good at systematically exploring all functionalities integrated in a single GUI layout [17]. If a user trace can lead to such a screen with abundant GUI elements, the test coverage can be significantly and systematically improved by automated testing (lots of existing work available for this purpose).
2) UGA can also select an event $e_k \in A$ as a stop point if $e_{k+1}.t - e_k.t$ is large enough (e.g., greater than a given threshold). The rationale behind this rule is that if there has been a large interval between two continuous events, it is very likely that the user creating this trace has spent much time on checking an app's GUI to explore next interactions for trying more functionalities. In such cases, applying automated testing can systematically traverse a large number of app states, and this task is instead not effective for a user to complete manually. To identify such stop points, UGA calculates the interval between each pair of two consecutive events (i.e., $e_{i+1}.t - e_i.t$). If the interval is significantly larger than those of other pairs, $e_i$ can be selected as a stop point.

### D. User-guided Automated Testing

Given an app's user trace $T = \{e_1, e_2, \ldots, e_n\}$ and an identified stop point $e_k$ ($k \leq n$), UGA replays the app until reaching this stop point (i.e., feeding events $e_1, e_2, \ldots, e_k$ to the app and observing its execution), and then switches to automated testing. For testing purposes, one may adopt various existing automated testing techniques, e.g., random testing, model-based testing, white-box testing, and so on. In this paper, we choose to adopt two representative testing techniques: random testing (RND) [8], [9] and model-based depth-first testing (DFS) [6], [11]. These two techniques are being widely used in practice. We adapt the RND idea to our problem and present it in Algorithm 1. RND would randomly generate and perform a sequence of interactions by analyzing the runtime GUI layout of an app under testing. Executing RND($v, \ell$) yields a $\ell$-step random exploration starting from activity $v$. DFS, on the other hand, treats an app as a state transition system (i.e., activities as states, and interactions as transitions between states), and aims to systematically traverse all states and cover all transitions. We also adapt the DFS idea to our problem and present it in Algorithm 2. We keep global variable $V$ to store the activities being visited. Executing DFS($v$) yields a systematic exploration of app states staring from activity $v$.

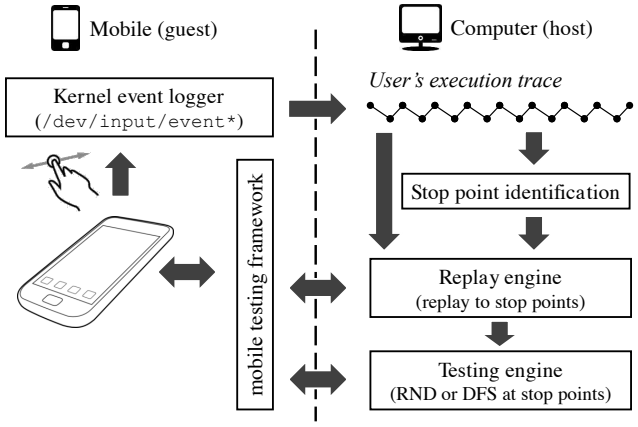



Figure 4: Overview of our UGA implementation

## IV. Implementation

We have implemented our UGA idea as a prototype tool for Android platform. In this section, we discuss implementation details about user trace recording, replaying, and built-in automated testing techniques. Figure 4 gives an overview of tool implementation. For ease of presentation, we also name our tool UGA.

Our UGA tool conducts user trace recording, replaying and automated testing on a real Android device (named *guest*), which is controlled by a computer (named *host*) via Android Debug Bridge (adb).

To record user traces, we modified RERAN [12], a non-intrusive record and replay tool for Android. Specifically, when an app runs on the guest device, the host computer will collect kernel input events through adb, pulling time-stamped data from `/dev/input/event*` device files using a "getevent" command. To replay user traces, the host computer sends its logged events to the guest device, and the latter feeds these events to the app under testing according to their occurring order until reaching a stop point.

Stop points are determined by the two heuristic rules discussed earlier in Section III. In order to achieve high test coverage, our UGA tool gives priority to stop points that are associated with longer time gaps or more GUI elements. In other words, UGA would prefer to conduct automated testing at stop points with higher priorities, if there is only limited budget on time or other resources.

To conduct automated testing at stop points, we customized Troyd [18], which is based on the widely-used Robotium [15] testing framework. Specifically, Troyd is a background service residing in the guest device, and can respond to the host computer's requests through a socket connection. Our automated testing is conducted in a three-step fashion. First, UGA uses Troyd to obtain the guest device's dynamic GUI layouts at runtime, and send their information back to the host computer. Second, the host computer decides which actions to execute according to a certain testing technique (e.g., Algorithm 1 or 2). Finally, the host computer sends these actions back to the guest device for execution, and Troyd will take over these execution tasks.

## V. Evaluation

In this section, we evaluate the effectiveness of our UGA technique experimentally with real-world Android apps.

### A. Experimental Setup

We conducted our experiments on a PC with Intel Q9550 CPU and 4GB RAM running Ubuntu 13.04, and a Google Nexus 4 smartphone running Android 4.4.2.

We selected seven real-world Android apps from two primary app distribution channels: Google Play [19] store and F-Droid [20]. These apps are being popularly downloaded from market and cover different application categories. Table I summarizes basic information about these apps, including their names, categories, number of downloads, number of activity components, number of methods, and app sizes in terms of the number of bytecode instructions.

To evaluate our UGA's effectiveness, we measured method coverage for each app when using the UGA implementation configured to RND and DFS algorithms, respectively, i.e., UGA+RND and UGA+DFS. The coverage data were collected by instrumenting each app's bytecode using apktool [21]. Higher method coverage implies potentially more thorough testing [6]. Since we test apps themselves, when measuring method coverage, we considered only code in the core package of each app and excluded all external libraries. In the following, we explain more setup details:

1) RND: UGA repeatedly fed each app with randomly generated input interactions with length $L = 10$, until reaching a time limit of 60 minutes. In our experiments, we found that further increasing $L$ or time limit only improve test coverage marginally.
2) DFS: UGA ran the depth-first exploration until its termination (i.e., all states and transitions in a GUI model are fully traversed). This process took less than 60 minutes for all our selected apps.
3) UGA+RND/UGA+DFS : UGA collected a user trace for each app by inviting a mobile phone user without any mobile app development experience. The user was asked to play the app in any way for at most 10 minutes. For each collected user trace, UGA selected 10 stop points according to our two heuristic rules discussed earlier in Section III, and conducted automated testing using RND or DFS from these stop points.

### B. Experimental Results

Table II presents our experimental results and comparisons between different configurations. Column 3 lists method coverage data that were achieved by replaying collected user traces only. Columns 4, 5, 7 and 8 list method coverage data achieved by RND only, UGA+RND, DFS only and UGA+DFS, respectively. As we can observe, method coverage is 13.5–45.8% (median 25.5%) for replaying user traces only, 3.2–44.5% (median 25.4%) for applying RND only and 3.2–50.8% (median 25.6%) for applying DFS only. These results are similar to those reported in existing work [7]. We thus observe that simply replaying user traces or applying state-of-the-art automated testing techniques like RND and DFS can only achieve very limited test coverage for Android apps. This is because users are not good at systematic exploration of a large state space, and existing automated testing techniques also have challenges when facing complex interactions. Nevertheless, when one combined them together, i.e., using UGA+RND or UGA+DFS, both of them achieved significantly improved test coverage. For example, UGA+RND improved method coverage by 14.6–43.8% (median 32.5%) against RND only; UGA+DFS also improved method coverage by 20.2–67.4% (median 30.8%) against DFS only. These results confirm that exploiting user traces can indeed boost the effectiveness of an automated testing tool.

We also observe that DFS performed slightly better than RND (improvement is in a range of 0–6.3%, with a median of 0.3%). This observation is consistent with findings reported in related studies. For example, researchers found that DFS can achieve the same test coverage faster than RND, but finally the two techniques will converge to similar coverage given enough time and testing resources [3], [11]. One interesting finding from our experiments is that UGA improved DFS much more than it did for RND in terms of test coverage. For example, even with the same user traces and stop points, UGA+DFS achieved significantly better coverage improvement than UGA+RND (in a range of 2.8–23.6% with a median of 6.4%). This finding further validates our intuition that user

Table I: Android apps used in our experiments

| *Name* | *Category* | *# Downloads*[2] | *# Activities* | *# Methods* | *# Instructions* |
|---|---|---|---|---|---|
| Amazon | shopping | 5M–10M | 3,264 | 18,431 | 706,645 |
| Any.do | to-do list | 5M–10M | 1,433 | 7,326 | 334,118 |
| Netease News | news client | 500K–1M | 1,609 | 10,806 | 507,438 |
| Mileage | car manager | 500K–1M | 221 | 1,185 | 51,979 |
| Tippy Tipper | tip calculator | 50K–100K | 56 | 238 | 10,138 |
| Alarm Klock | alarm clock | 500K–1M | 160 | 673 | 30,027 |
| Bing Dictionary | dictionary | 10K–50K | 581 | 2,374 | 137,592 |

Table II: Evaluation results

| *Subject* | *#Methods* | *Method Coverage* | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | *User trace only* | *RND only* | *UGA+RND* | $\Delta$ *vs. RND* | *DFS only* | *UGA+DFS* | $\Delta$ *vs. DFS* |
| Amazon | 18,431 | 21.9% | 25.4% | 42.5% | 17.1% (1.67×) | 25.6% | 47.6% | 22.0% (1.86×) |
| Any.do | 7,326 | 13.6% | 10.8% | 25.4% | 14.6% (2.36×) | 11.7% | 31.8% | 20.2% (2.73×) |
| Netease News | 10,806 | 25.5% | 8.7% | 49.2% | 40.5% (5.65×) | 9.0% | 53.5% | 44.5% (5.94×) |
| Mileage | 1,185 | 33.4% | 27.3% | 62.4% | 35.1% (2.29×) | 27.3% | 79.3% | 52.1% (2.91×) |
| Tippy Tipper | 238 | 45.8% | 44.5% | 72.3% | 27.7% (1.62×) | 50.8% | 80.7% | 29.8% (1.59×) |
| Alarm Klock | 673 | 39.2% | 41.2% | 73.7% | 32.5% (1.79×) | 45.8% | 76.5% | 30.8% (1.67×) |
| Bing Dictionary | 2,374 | 13.5% | 3.2% | 47.1% | 43.8% (14.51×) | 3.2% | 70.6% | 67.4% (21.78×) |

traces can provide a sketch of an app's state space, and this enables the remaining state space to be explored by automated testing techniques.

### C. Discussions

We now look into the insights why our UGA technique is effective in improving test coverage for mobile apps.

There can be many factors that limit the effectiveness of an automated testing technique. Examples include that: (1) complex interactions can prevent automated testing from reaching a specific app state; (2) deep functions (those functions that can only be invoked by a long input sequence) are difficult to reach by random testing; (3) imprecise model abstraction can omit part of an app's state space to be explored. However, user traces usually contain hints on addressing such challenges. In the following, we discuss three scenarios encountered in our experiments, in which existing automated testing techniques failed to cover apps adequately, while our UGA significantly improved test coverage with the help of collected user traces.

*Scenario 1* (*Bing Dictionary*). When the Bing Dictionary app was launched for the first time, it required three swipe gestures to skip its starting screens and reach its main screen. Existing automated testing cannot easily generate corresponding complex interactions to achieve this task[3]. As a comparison, our UGA can simply replay collected user traces to pass the starting screen as well as other difficult-to-pass screens, and systematically explore remaining GUI elements, resulting in significantly improved test coverage.

*Scenario 2* (*Mileage*). Mileage is a car managing app. Some of its functionalities can only be enabled after its user has customized a vehicle. However, the customization needs a long sequence of input actions. It is therefore difficult for RND and DFS to reach and test such functionalities. As a comparison, users can easily customize their cars themselves when using this app. Therefore, UGA can replay collected user traces to easily enable such functionalities for later thorough testing.

*Scenario 3* (*Netease News*). The third example is the Netease News app. It has a parameter setting screen, which can only be reached by a certain swipe gesture. All existing automated testing technique tried in our experiments failed to reach this screen, because related GUI layout models have no explicit transition from the app's main screen to this parameter setting screen. However, our UGA enhanced GUI layout models by recorded user traces, and succeeded in reaching this setting screen as well as exploring more code from this screen. Besides, this was done in an automated way.

## VI. Related Work

Modern software applications mostly contain GUIs. To automatically test such applications, researchers have proposed various model-based techniques [22]–[24]. For example, GUITAR [22] is a well-known systematic GUI testing technique. Given an application, GUITAR extracts a state-machine based GUI model from this application, and tries to traverse all its states (i.e., windows) and all transitions between these states (i.e., clickable widgets on windows) for exposing bugs.

Many mobile apps are also GUI applications. Thus model-based GUI testing also applies to such apps [5]–[11], [13], [25]. For example, one can model an app's states by means of screens [6] or enabled GUI elements [7], and model transitions by means of valid actions on these enabled GUI elements. Such GUI information can be statically derived from an app's source code [11], or dynamically obtained at runtime [6], [7]. With such modeling, one can extract a state-machine based GUI model for each mobile app. Then different testing

[2]1K=1,000 and 1M=1,000,000. Also, we count only number of downloads from Google Play store as it is official. Our selected apps are also being popularly downloaded at other third-party markets.

[3]It is still possible for automated testing techniques to generate certain gestures [6], but that would be inefficient. This is because it is generally unknown how to predict which apps need what gestures as inputs. A tool would have to generate a large set of common gestures configured with various parameters (e.g., start and end points of a swipe gesture) for testing, and most of these gestures would be totally irrelevant.

techniques or strategies can be applied on these extracted GUI models. Typical examples include depth-first search [6], [11], heuristic-rule based search [7], [25], random exploration [8], [9], [13], and symbolic or concolic analysis [5], [10].

Despite many existing automated testing and GUI model extraction tools (e.g., Monkey [26], Robotium [15] and Troyd [18]), how to effectively synthesize complex interactions (e.g., drawing an unlock pattern on screen in our motivating example) still remains to be an open and emergent question. In practice, these pieces of existing work are still incapable of automatically generating complex interactions to effectively test a mobile app. Instead of blindly pursuing test automation, our UGA technique takes one step back and addresses this problem by exploiting user trace information to support effective mobile app testing. Besides, [6] also proposed a targeted exploration technique to more thoroughly explore an app's state space. It works by forcing activity launches according to the statically generated activity transition graph. It shares exactly the same key insight with UGA: leveraging additional information to explore the "hard-to-reach" states, and we innovatively addressed this issue by utilizing information contained in the user traces.

On the other hand, there is also existing work on human-assisted testing for GUI applications. For instance, Amalfitano et al. [27], [28] proposed to extract GUI models from collected user traces for testing, and Huang et al. [29] demonstrated how to manually identify and prioritize transitions in GUI models for testing. Compared with these pieces of work, our UGA requires only very limited user traces (e.g., may only one), which can be easily obtained with negligible manual effort. Besides, UGA does not require its users to have special mobile testing skills, and can thus be easily deployed in a crowdsourcing way.

Finally, from an even broader perspective, researchers have kept showing that challenging problems for computers (not only testing) can be alleviated or solved in a human-assisted way [30]–[32]. For example, Chen et al. [30] demonstrated that it is possible to decompose complex problems into small puzzles for humans to solve, and this semi-automated solution can achieve a significant improvement over a fully automated approach. Our UGA shares a similar insight with these approaches, and our evaluation confirmed that it is indeed a successful attempt towards introducing human assistance to better support the testing of mobile apps.

## VII. Conclusion

In this paper, we proposed a novel user-guided testing technique for mobile apps, targeting at improving test effectiveness. To the best of our knowledge, this is the first attempt that integrates user insights into automated testing techniques in a cost-effective way. Our experiments on seven real-world Android apps show that our UGA technique can achieve a significant improvement over existing automated testing techniques in terms of code coverage. Based on these promising results, we believe that our research sets a new direction towards cost-effective testing for mobile apps.

Still, our work is at an initial stage. There are many research issues that deserve further study. UGA is more a framework than a single technique. Every different user trace collection strategy, stop point identification algorithm and automated testing approach defines an unique UGA implementation: UGA defines a spectrum of mobile testing techniques, and more research issues are emerging. It could be argued that our UGA's test effectiveness might depend on what user traces are collected and how stop points are identified. Then how to guide users to provide useful traces and select quality stop points need a further study. Also, we observed that UGA achieves higher coverage improvement for model-based systematic testing (e.g., DFS), as compared with random testing. Theoretically, any automated testing techniques can be extended for such user guidance. Then we wonder: should we guide users in different ways when different underlying automated testing techniques are deployed? How such customization can guarantee more improvement in test effectiveness? We are working along this line.

## Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472177, 91318301, 61321491, 61361120097) of China.

## References

[1] S. Allen, V. Graupera, and L. Lundrigan, "The smartphone is the new PC," in *Pro Smartphone Cross-Platform Development*. Apress, 2010, pp. 1–14.

[2] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell, "A survey of mobile phone sensing," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 140–150, 2010.

[3] W. Choi, "Automated testing of graphical user interfaces: A new algorithm and challenges," in *Proceedings of the 2013 ACM Workshop on Mobile Development Lifecycle*, ser. MobileDeLi, 2013, pp. 27–28.

[4] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE, 2014, pp. 1013–1024.

[5] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE, 2012, pp. 59:1–59:11.

[6] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications*, ser. OOPSLA, 2013, pp. 641–660.

[7] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA, 2013, pp. 623–640.

[8] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST, 2011, pp. 77–83.

[9] ——, "A GUI bug finding framework for Android applications," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC. ACM, 2011, pp. 1490–1491.

[10] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[11] W. Yang, M. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7793, pp. 250–265.

[12] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and touch-sensitive record and replay for Android," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 72–81.

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

[14] "Alipay Wallet App," https://play.google.com/store/apps/details?id=com.eg.android.AlipayGphone.

[15] "Robotium Tool," https://code.google.com/p/robotium.

[16] "Hierarchy Viewer," http://developer.android.com/tools/help/hierarchy-viewer.html.

[17] G. Bae, G. Rothermel, and D.-H. Bae, "On the relative strengths of model-based and dynamic event extraction-based GUI testing techniques: An empirical study," in *Proceedings of the 23rd International Symposium on Software Reliability Engineering*, ser. ISSRE, 2012, pp. 181–190.

[18] J. Jeon and J. S. Foster, "Troyd: Integration testing for Android," *Technical Report CS-TR-5013*, 2012.

[19] "Google Play Website," https://play.google.com/store.

[20] "F-Droid Free and Open Source App Market Website," https://f-droid.org.

[21] "Apktool Tool," https://code.google.com/p/android-apktool.

[22] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," *Proceedings of the 20th Working Conference on Reverse Engineering*, vol. 0, p. 260, 2003.

[23] A. M. Memon, "An event-flow model of GUI-based applications for testing: Research articles," *Softw. Test. Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, 2007.

[24] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65–105, 2014.

[25] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY, 2013, pp. 209–220.

[26] "Monkey Tool," http://developer.android.com/tools/help/monkey.html.

[27] D. Amalfitano, A. Fasolino, and P. Tramontana, "Reverse engineering finite state machines from rich internet applications," in *Proceedings of the 15th Working Conference on Reverse Engineering*, ser. WCRE, 2008, pp. 69–73.

[28] ——, "Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM, 2009, pp. 571–574.

[29] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang, "Design and analysis of GUI test-case prioritization using weight-based methods," *Journal of Systems and Software*, vol. 83, no. 4, pp. 646–659, 2010.

[30] N. Chen and S. Kim, "Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE, 2012, pp. 140–149.

[31] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum, "reCAPTCHA: Human-based character recognition via web security measures," *Science*, vol. 321, no. 5895, pp. 1465–1468, 2008.

[32] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop, "Pex4Fun: Teaching and learning computer science via social gaming," in *Proceedings of the IEEE 25th Conference on Software Engineering Education and Training*, ser. CSEET, 2012, pp. 90–91.