Facilitating Reusable and Scalable Automated Testing and Analysis for Android Apps

Zhanshuai Meng, Yanyan Jiang, Chang Xu*
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
mzsnanju@gmail.com, jiangyy@outlook.com, changxu@nju.edu.cn

ABSTRACT

Mobile apps are prevalent in everyone's daily life. However, apps are oftentimes defective, undermining their convenience, and therefore automated testing and analysis of apps are developed to enhance apps' quality. As these advanced technologies become increasingly effective and complicated, prototyping such a tool also becomes a challenge. To facilitate easy development of high-quality testing and analysis tools that work with real-world apps, we in this paper present the design and implementation of ATT (Android Testing Toolkit), for crafting reusable and scalable testing and analysis on Android apps. ATT consists of integrated tools and APIs for event generation, profiling and program instrumentation, and can be distributed over cloud platforms for scalable testing and analysis. We qualitatively evaluated the applicability of ATT by demonstrating implementation of a series of existing and enhanced work (RERAN, Monkey+ and UGA) upon ATT, and quantitatively studied the scalability of parallelized UGA on a cloud platform with five real-world apps. We believe that our ATT tool would facilitate development of more advanced testing and analysis approaches for Android apps.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Design, Reliability, Experimentation

Keywords

Android, testing, framework

* Corresponding author.

Internetware '15, November 6, 2015, Wuhan, China
© 2015 ACM. ISBN 978-1-4503-3641-3/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2875913.2875915

1. INTRODUCTION

Mobile apps are playing an increasingly important role in our daily life. Rich interactive capabilities (e.g., gestures and sensors) of mobile devices facilitate a wide range of apps to be developed, ranging from entertainments to productivity suites.

Development of mobile apps are made easier by carefully designed SDKs, rich documentation and community support, and therefore the mobile app market is flourishing in a remarkable rate. However, mobile apps are oftentimes not thoroughly tested due to the requirement of timeliness releases, and thereby app releases are frequently defective, causing troubles for the end-users, which offsets the convenience the app brings, or even causes more severe security issues [26].

To improve app quality, automated testing approaches are extensively studied. Automated testing techniques target generating input sequences that can better cover the state space of an app, detecting defects early before the app is released. An automated testing technique can be random-based [7, 13, 14, 17], model-based [9, 10, 11, 19, 20, 21, 22, 29], or with human intelligence [16]. Advanced analyses and heuristics are also carried out to enhance the effectiveness and efficiency of the automated testing procedure. For example, adversarial events can be generated to detect specific categories of defects [15], and symbolic execution or search-based code analysis can be applied in systematic event sequence generation [8, 23].

Though being extensively evaluated and validated, not all these tools (usually a prototype implementation) can be easily realized for use. Monkey [7] is the most widely-adopted testing tool that meets industrial standard. Monkey is efficient for unit testing (e.g., testing a single widget or activity). However, it has limited coverage for systematic testing because the probability of reaching a deep event is exponentially small. More advanced testing approaches [8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 29] are proposed. However, they relatively fall short in usability for an inexperienced developer. They oftentimes (1) rely on unstable analysis tools; (2) only support a subset of real-world apps' features, and are implemented of proof-of-concept prototypes. For example, SwiftHand [11] adopts a dynamic finite state machine to systematically explore the state space of an app. However, it only generates scrolling and touching UI events and does not take into account system events (e.g., lifecycle events or sensory events) or even more complex UI events, which are common in real-world apps.

Such limitations cause problem realizing these approaches to embrace the practical aspects of real-world apps, as extending them with application-specific functions would be labor-intensive: even extending the simplest Monkey random testing suit requires knowledge of a large code base, not to mention the research prototypes.

To bridge the gap between the research proposals and the practical need of testing tool implementation, we in this paper propose a testing framework called ATT (Android Testing Toolkit) for Android, the most prevalent mobile platform, which offers a set of APIs to facilitate testing tool construction and Android app analysis. We extracted the basic needs of implementing testing tools, roughly categorized as follows:

- Device management, for allocation and recycling virtual machines and testbed devices;
- Event generation, for injecting UI, sensory and system events to a test device;
- System profiling, for monitoring GUI layout and system states;
- Program instrumentation, for manifesting more advanced customized analyses (e.g., test coverage measuring).

We implemented ATT in Java upon adb [1], Robotium [24] and Android instrumentation framework [6]. With ATT, developers can focus on the actual testing-related code with a high-level event model, and ATT takes care of the rest device-dependent details. Finally, ATT is capable of transparently managing a cluster of Android devices or emulators, enabling automated massive parallel testing.

To evaluate the applicability of ATT, we demonstrate ATT implementation of Monkey+, an enhanced Monkey-like random testing tool, and user guided automation (UGA) [16]. We also extended the original UGA to embrace the parallel testing function of ATT, and conducted efficiency study on a cloud platform. The results show that parallel testing of ATT scaled well, and can cover much more code than single-threaded UGA within a given time budget.

In summary, the main contributions of this paper are listed as follows:

- We propose an Android testing framework ATT which provides a set of APIs for the input generation, program instrumentation and testing system control and monitoring, as well as massive parallel testing support. With the infrastructure provided by ATT, developers can conveniently implement existing testing techniques.
- We demonstrate the power of ATT by implementing the extended random testing tool Monkey+, which inherits some basic features of Monkey and has more powerful features to facilitate heuristic random testing. Moreover, we evaluated the scalability of parallel version of UGA implemented on ATT. The results confirm that ATT is capable of effectively controlling and monitoring multiple emulators.

The rest of this paper is organized as follows. Section 2 presents the overall architecture of our system. Section 3 describes the design of APIs in ATT. Section 4 describes our implementation. Section 5 presents our experimental results. Section 6 discusses related work and finally section 7 concludes.

2. SYSTEM OVERVIEW

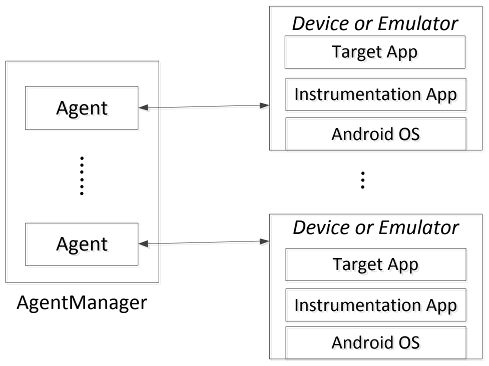

Figure 1: ATT architecture

In this section, we present an overview of ATT. For illustration, we use a motivating example to discuss the limitations of some existing techniques (Section 2.1) and the architecture of our work (Section 2.2). We then explain how our ATT framework works (Section 2.3).

2.1 A Motivating Example

RERAN [12] is a non-intrusive record and replay tool for Android. RERAN directly captures and replays the low-level events that are triggered on the device. In RERAN, the replay agent is wrote in C and runs on the device. RERAN serves as a basis of our previous work UGA [16] that leverages the human insights in testing mobile apps. However, RERAN is only capable of replaying a trace, while UGA requires further analysis of it, and hence we had to implement a separate trace analysis tool from scratch. This motivates our design and implement of ATT. ATT allows us to implement both event record and trace analysis in a single pass.

2.2 System Architecture

Figure 1 presents the system architecture of ATT. Given an app, ATT is capable of (1) providing a rich set of API for program instrumentation, testing and analysis; (2) automatically scheduling physical mobile devices or emulator instances for running analysis code.

ATT is implemented based on a master-slave model. The coordinator named AgentManager runs on a master JVM and it is capable of interacting with multiple slave Agent nodes. Each agent runs separately and manages a single Android device or emulator. It receives the dynamic state information from a background service (named instrumentation app) and sends commands to the instrumentation app to perform actions on the target app.

Agents run on the computer (named host) and connect to real devices or emulators on the cloud server via adb tool [1]. The agent instance initializes a device by installing both the target app and the instrumentation app to the device or emulator. The background instrumentation service which runs in the same process as target app then connects to the agent running on the host in return. Each agent obtains the GUI hierarchy and current activity of app under test through the instrumentation app dynamically. And it is capable of monitoring the app's state, collecting available actions for the current activity and performing them via the background instrumentation app.

AgentManager object monitors and controls all of the living agent instances. Agents can be created on demand as long as there are sufficient resources in the cloud system. Testing and analysis is conducted on every standalone agent separately, facilitating various testing techniques to be executed on disparate devices or emulators in parallel. AgentManager is responsible for the primary management work, and it periodically collects profiling information from all running agents.

2.3 ATT Workflow

The following example in Java shows how to use ATT to implement the replay technique on the basis of ATT APIs.

```
...
AgentManeger am = new AgentManager();
//create agent instances
Agent agent = am.createNewAgent();
//get device or emualtor started
agent.startNewDevice();
//collect the information of apk file
ApkHandler aut = new ApkHandler(new File(
    apkPath));
//instrument, modify, repack and sign.
agent.loadApp(aut);
agent.startUpApp(aut);
...
//find and tag all the stop points
List<event> userSPTrace = agent.findAllSP(
    userTrace);
//replay
for(int i=0;i<userSPTrace.size();i++) {
  agent.fireEvent((Event)userSPTrace.get(i
      ));
  if(i>0)
    agent.wait(userSPTrace.get(i).
        getTimeStamp()-userSPTrace.get(i
        -1).getTimeStamp());
  if(userSPTrace.get(i).isSP())
    break;

}
...
```

Listing 1: UGA Replay by ATT

In the above code snippet, the single AgentManager object `am` is responsible for simultaneously controlling and monitoring multiple agents. An Agent object named agent are created by calling `am.createNewAgent()`. An available Android device is initialized by calling `agent.startNewDevice()`. The followed three lines help us instrument, sign the target app and launch it finally. All the stop points of the logged user trace are identified by calling `agent.findAllSP(userTrace)`. And then, we replayed the recorded user trace to the identified stop points.

ATT provides rich APIs for the developing of testing tools. Furthermore, it is capable of testing in parallel on a cluster of multiple standalone agents.

2.4 Testing Support

2.4.1 Parallel Testing

Parallel testing means testing multiple applications or subcomponents of one application concurrently to reduce the test time. Parallel tests consist of two or more parts that check different parts or functional characteristics of an application. Our ATT is capable of conducting the testing process in parallel under the cooperation of the AgentManager instance and the multiple standalone agents. The GUI hierarchy of application is fetched dynamically by every single agent. Dynamic analysis of the GUI components helps to collect available actions to perform. AgentManager monitors and controls all of agent instances and obtain the statistical information via adb [1].

3. API DESIGN

ATT is proposed to facilitate testing tool construction and Android app analysis and it provides a rich set of API in our design. As described in Section 1, ATT is supposed to meet the basic extracted needs for implementing testing tools. In the following subsections, we discuss the overall design of categorized APIs in sequence.

3.1 Deployment

ATT is capable of transparently managing a cluster of Android devices or emulators automatically. All of the running devices are supposed to keep from tripping over each other. It is assumed that ATT controls and recycles the multiple devices or emulators efficiently. And in the architecture of ATT, multi-components connect with each other through socket communication during running, which requires that the system offer efficient automatic management of port assignments. Additionally, agents are capable of transferring or sharing files with the device, such as the installation of target app and obtaining the device configuration (e.g., the resolution of device). Therefore, ATT provides the set of APIs categorized as deployment here.

3.2 Event Generation

As it is known, Android app is event-driven, and inputs at the basic level are in the form of events, which contain either classified UI events, such as touchscreen, scrolls and text inputs, or system events, such as the notification of low battery level and lifecycle events. ATT handles both UI events and a portion of non-UI events because significant functionality of mobile apps is controlled by some system events. A couple of UI input mechanisms are supported in our design, including user screen touch and navigation button press (further, “back”, “home” and “menu” button presses).

ATT also generates system events on the fly. For instance, an navigation app is supposed to respond to the change of GPS location in a proper manner. System events are derived from either the broadcast receiver or the system service. An app registers a callback dynamically or statically, which will be invoked as system events.

3.3 System Profiling

During the tedsting process, agents are supposed to monitor and obtain the system states faithfully. For example, each agent should be aware of whether the target app is running normally or has been switched to the background or stopped already. And each agent obtains the GUI hierarchy and current activity during the testing. We designed the AndroidView class to address the representation of GUI hierarchy in ATT as Table 1 shows. A list of AndroidView objects which are displayed in the focused activity or dialog denote the current GUI hierarchy structure. We explicitly designed ATT as a dynamic tool so that it can obtain the current layout timely and is transparent to the app. ATT leverages Java's reflection mechanism to get the list of Android widgets of an activity, and therefore correctly captures the GUI hierarchy even if it is updated by an asynchronous tasks in background.

In our design, ATT registers handlers which listen to the injection of events and will be invoked once the injection happens. Moreover, it does not require access to app source code, perform any app rewriting, or perform any modifications to the virtual machine or Android platform. We implemented RERAN (Record and Replay), which can help to reproduce the detected bugs during testing, based on the callback technique in Section 5.

3.4 Apk Handler

Some Android apps depend on the required libraries on configurations to get started. And generation of input may need the static information specified in `AndroidManifest.xml`, such as the broadcast receivers' information. For this reason, we designed the apk handler in charge of extracting the static information of target apk files into specified data structure.

3.5 Program Instrumentation

ATT allows users to manifest more advanced customized analyses (e.g., test coverage measuring) and monitor the interaction the system. For example, ATT is capable of adding a new monitor that will be checked whenever an activity is started.

4. SYSTEM IMPLEMENTATION

This section describes the implementation details of ATT. We start by introducing the instrumentation app (Section 4.1), and then discuss the API implementation (Section 4.2) and supported testing (Section 2.4).

4.1 Instrumentation App

We implemented the instrumentation app built on the high-level APIs supplied by Robotium [24], Android instrumentation framework [6] and adb tool [1]. Android Instrumentation framework allows us to monitor all of the interaction the system has with the application and to inject events. Furthermore, Robotium provides abundant APIs layered on top of Java reflection and instrumentation framework.

Instrumentation app which runs as a background service on the mobile device, shares the same process as the target app. To test and analyze the app, agent sends commands to the instrumentation app for performing action or collecting data. Agent sets up socket communications with the instrumentation app and obtains the list of widgets of current activity and then dynamically analyzes the GUI components of displayed screen to collect available actions.

4.2 API Implementation

In the following subsections, we discuss the implementation details about system controlling and monitoring, program handling, and event generating.

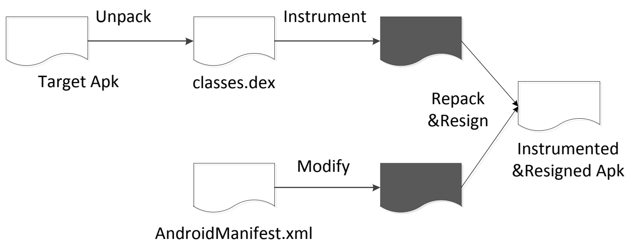

Figure 2: App modification process

4.2.1 *App Instrumentation, Repacking and Resigning*

Dominant Android platform requires that the instrumentation app and target app be signed by the same key sharing the same process space. Moreover, agent is supposed to be instantiated and communicate with the instrumentation app through socket, which requires the latter have network permission. We use ApkTool [3] to unpack the target app and modify the obtained `AndroidManifest.xml` file by adding multiple corresponding tags to it. Afterwards, the target app is repacked and resigned for the coming testing as figure 2 shows.

4.2.2 *App Information Extraction*

ATT extracts the statically specified information of target app into an ApkHandler object before testing. To do this, We get the `AndroidManifest.xml` with the target app unpacked via ApkTool [3]. ATT processes `XML` file to find necessary information of target app, including launcher activity, library dependencies and all of the activities, services and broadcast receivers specified statically in the running app under test, which may be useful for the generation of input.

4.2.3 *Event Generation*

As for unalloyed implementations of UI event generation, Robotium [24] plays a vital role. The robotium-rooted instrumentation app assists in capturing the GUI widgets and actual information content of the layouts, such as the coordinates and identifiers of view widgets. UI events are generated using the adb tool [1]. We set the fixed keycode value as the arguments of adb [1] to generate the navigation button press events.

The Android SDK [2] has a variety of system events as intents, to which the app may response by registering of broadcast receivers. The identification of valid values of the data associated with the intent is crucial. We take the statically registered broadcast receivers into account, which are specified in `AndroidManifest.xml`. We choose the intents which do without any data object to handle. And as to the generation of specified data, we will tackle with it in our future work.

The target app may register or unregister listeners for the global system services. For instance, once the GPS status changes, the specified registered listener receives a corresponding system event. We mock the modification of data belonging to a couple of general system services, Location-

Table 1: AndroidView fields

Modifier	*Type*	*Field*	*Description*
`private`	`int`	`id`	specified id in resource files
`private`	`Coordinate<float,float>`	`coordinate`	relative coordinates
`private`	`int`	`width`	obtained width
`private`	`int`	`height`	obtained height
`private`	`AndroidView`	`topParent`	absolute top parent view of the specified widget
`private`	`List<AndroidView>`	`children`	list of views located under the specified widget

Table 2: UI event,l,t,w,h denote left position,top position,width,and height of the view.

Category	*Event*	*adb shell commands*	*Description*
screen touch	Tap	input tap $l+w/2$ $t+h/2$	tap at center of view
	LongTap	input swipe $l+w/2$ $t+h/2$ $+w/2$ $t+h/2$	LongTap at center of view
	Drag	input swipe $l_1+w_1/2$ $t_1+h_1/2$ $l_2+w_2/2$ $t_2+h_2/2$	swipe from view$_1$ to view$_2$
	Text	input text string	trigger test input
navigation button press	PressBack	input keyevent KEYCODE_BACK	press back button
	PressHome	input keyevent KEYCODE_HOME	press home button
	PressMenu	input keyevent KEYCODE_MENU	press menu button

Manager and SensorManager to generate the specified system events by sending telnet commands to the emulator.

Rotating means that the orientation of the device is changed by users. ATT generates this event with the help of Robotium. The user rotates the device and rotates it back after a while, which causes the current activity is destroyed and recreated in turn. And this frequently occurs in practice. In addition, Android instrumentation framework [6] gives ATT a favor in the generating of lifecycle events.

Table 2 and 3 presents the implementations of UI events and system events in ATT.

4.3 Limitations

Currently, the bytecode instrumentation of some apps may result in failure when launched by ATT. And our dependent ApkTool [3] suffers for specified real-world apps (e.g., an app containing integrity verification).

ATT sometimes becomes out of work when it comes to the custom widgets which depend on the complex interval dispatch logic. And it does not support the apps whose entry routing is native and it is unable of controlling the internal native part the apps. This is because that our instrumentation can only deal with the Java-like bytecode. Both of the aforementioned can lead to incomplete information obtained during the program execution, especially for apps that heavily depend on the native libraries (e.g, games).

5. EVALUATION

To evaluate our ATT tool, we first conduct qualitative study of applicability by demonstrating how ATT provides testing support for implementing Monkey+, RERAN [12] and UGA [16] in Section 5.1. We also conducted a quantitative study of the ATT's scalability by deploying parallel implementation of UGA on a cloud platform in Section 5.2.

5.1 Applicability

5.1.1 Monkey+

Monkey [7] is a program that automatically injects pseudo-random streams of user events such as clicks, touches, or gestures runs for Android apps. Monkey is one of the most prevalent Android testing tool and is officially included in the SDK. However, the efficiency of Monkey oftentimes cannot meet developer's need because of the following limitations.

- Monkey uses brute-force random strategy and touches random coordinates on the screen. Most times such events touch nothing (the coordinate even might not been associated with an event handler) and time is simply wasted. There is no way to guide Monkey to get aware of the GUI layout unless we modify its source code, which would certainly be tedious.
- Monkey considers the target app as black-box and can only generate UI events. It cannot support any system events or complex UI gestures.
- Monkey is unable to generate UI events associated with structured data. For example, it would be challenging for Monkey to typeset a username and password combination.
- Monkey runs in the emulator or device environment, and it must be launched from a shell in that environment. This means that it can neither send user events to multiple systems of Android devices concurrently, nor coordinate between parallel testing clients.
- Monkey cannot generate on-demand coverage report, which is useful in monitoring the testing procedure.

To demonstrate the applicability of ATT, we implemented Monkey+ in Java with ATT APIs. Monkey+ contains most basic features of Monkey (with the same command-line interface), plus an enhanced set of testing options. Table 4 shows all options supported by Monkey+. These highly customized additional options are easily implemented with the aid of ATT APIs. The entire implementation of Monkey+ is only approximately 900 lines of Java code, demonstrating that ATT is helpful in crafting new testing tools.

5.1.2 RERAN

The second tool to be implemented is RERAN [12], a record-and-replay tool for Android platform. It is capable of non-intrusively and faithfully logging of all low-level events

Table 3: System events

Category	*Action Name*	*Trigger Mechanism*
Broadcast Receiver Events	BATTERY_CHANGED	ActivityManager tool
	BATTERY_LOW	
	BATTERY_OKAY	
	TIME_SET	
	AUDIO_BECOMING_NOISY	
	DATE_CHANGED	
	USER_PRESENT	
	MEDIA_EJECT	
	BOOT_COMPLETED	
System Service Events	Location Change	telnet command "geo fix G",G is fixed geo-location
	Sensor Change	telnet command "snsor set S x:y:z"
Lifecycle Events	calling of an activity's lifecycle event handlers. (e.g.,callOnPause,callOnStop and callOnRestart etc.)	Andoird instrumentation frmaework
Rotate	change of the device orientation	Robotium

Table 4: Monkey+ command-line options reference

Category	*Option*	*Description*
Monkey and Monkey+	-s <seed>	Seed value for pseudo-random number generator.
	-n <num>	Set the number of event sequence generated.
	–throttle <milliseconds>	Insert a fixed delay between events.
	–pct-touch <percent>	Adjust percentage of touch events.
	–pct-nav <percent>	Adjust percentage of "major" navigation events. (Such as the back key, or the menu key.)
	-l <num>	Set the number of events generate.
Monkey+ only	-t text	Set the given text to any editable widget selected.
	-m <boolean>	Support multiple devices/emulators or not, true for yes.
	-setmode <boolean>	Switch the random strategy between the widget-based or coordinate-based, true for the former.
	–pct-sys <percent>	Adjust percentage of system events.
	-t <milliseconds>	Fix the delay between coverage report

that are triggered on the device, and such log can be used later for a replay of the past app execution.

We re-implemented the RERAN tool on the foundation of ATT. ATT allows developer to register event listeners on every event happened in the app. This functionality is perfectly suitable for implementing the record phase in RERAN. We register a listener on each user interaction event, and that event's corresponding parameters are logged and saved to a specified file. Following events are captured by our example RERAN implementation:

- Screen touch: tap, long tap, double tap, swipe;
- Navigation button press: press back, press menu and press home.

We developed an Android remote desktop application on Android version 4.4 or higher versions intently for the purpose of recording. Our remote desktop application serves as a VNC client connecting with the VNC server running on the emulators. It intercepts and recognizes the UI gestures that users conduct on the screen and then generates corresponding ATT events associated with type and coordinate information and sends them to the host via socket communications indeed. Afterwards, the received actions are performed with the help of APIs implemented in ATT. This remote desktop application is compatible with different resolutions of the multiple device screens.

For each event, the log entry is defined as a tuple $e = (c, t, s, a, g)$, indicating that event t is fired on the point with coordinate c at the time s and after that, the activity a is launched and a set of GUI components g are displayed on the current screen. For record-and-replay, providing (c, t, s) is sufficient. We also keep a and g because such information is required by further trace analyses, which is utilized in our UGA implementation introduced in Section 5.1.3. Replay is implemented by sequentially parsing the log file, and fire logged events at the correct timing. Our RERAN implementation contains only 88 lines of code.

5.1.3 UGA (RERAN and Monkey+)

Finally, we demonstrate the applicability of ATT by re-implementing our previous work: user guided automation testing (UGA) [16]. This final example is a combination of Monkey+ and RERAN. UGA leverages user insights to complement automated testing techniques by recording user-guided app executions, replaying apps to certain stop points and exploring state space from the chosen stop point to achieve maximum code coverage.

The first key part of UGA is observing human user's interaction with the app. RERAN is suitable for such task, and we use previously implemented RERAN to collect user trace, and conduct stop point analysis mentioned in the original paper. A stop point denotes a program state that has potential to be amplified by automated testing techniques.

The second key part of UGA is replaying the app to a stop point and conduct automated testing to amplify the user knowledge. We adopt the RND strategy in the original paper, as shown in Algorithm 1. We use our Moneky+ implementation as the RND algorithm.

Our UGA implementation is surprisingly small in footprint: it contains only 96 lines of Java code, if our previous Monkey+ and RERAN implementations are not counted. Since implementing testing tools are greatly eased by ATT, we believe ATT would facilitate rapid development of more advanced testing techniques.

```
Input: $\ell$ the length of generated events
function RND($\ell$)
begin
    for $i \in \{1, 2, \dots, \ell\}$ do
        $\mu$←getCurrentViewSet();
        $A$←$getAllEnabledActions(\mu)$;
        if $A \neq \varnothing$ then
            $a \leftarrow$ randomChoose($A$);
            execute $a$ ;
        end
        else
            break;
        end
    end
end
```

Algorithm 1: RND Algorithm in UGA

5.2 Scalability

Finally, we demonstrate that a parallelized version of UGA (PUGA) can also be easily implemented on ATT. Furthermore, we also qualitatively evaluated the effectiveness and scalability of PUGA on five real-world Android applications from F-Droid [4] and Google Play [5] stores. In PUGA, we assume that the target app is predominantly associated with the user interface, which means that we focus on automated user interface testing for Android apps in the implementation.

5.2.1 PUGA

Intuitively, a user trace would contain multiple stop points, and each stop point can be utilized for improving test thoroughness. Automated testing on each stop points can thus be fully parallelized to achieve even higher coverage in the given time budget. This motivates us to implement the PUGA technique.

PUGA has a two-phase design: a serial analysis phase, and a parallel testing phase. The analysis phase works exactly the same as UGA: a user trace is logged and stop points are extracted from the user trace. Suppose that user trace $T = \{e_1, e_2, \dots, e_{i_1}, \dots, e_{i_2}, e_{i_3}, \dots, e_n\}$ has been recorded, and $e_{i_1}, e_{i_2} \dots$ are identified as stop points. In the second phase, Our PUGA will start three parallel threads, and each thread creates an Agent class instances. Each agent a_j replays the user trace until its corresponding point e_{i_j} is reached, and hands over the execution control to Moneky+. Each agent would initialize an Android emulator on the fly, and agents' emulators are run fully in parallel. Moreover, since the agent threads are created in the same JVM, they can also be cooperative in the testing procedure. PUGA is obtained by a slight modification of our UGA implementation.

5.2.2 Experimental Setup

We evaluate the scalability of PUGA on a set of real-world programs from F-droid and Google Play store, as listed in Table 5. #Instructions, #Activities, #Methods columns exhibit number of bytecode instructions, activities and methods (with libraries excluded), respectively. The coverage data were collected by instrumenting each app's bytecode.

For collecting the user trace, we use our RERAN implementation described in Section 5.1.2. As did in our original paper, for user trace collection, the users are not needed to make themselves master of the target app and they only perform a simple and casual use of the app for a few minutes. Once user traces are collected, PUGA requires no more human participation.

For scalability study, we scale the parallel testing agents to be 1, 2, 4, 8, 16 and 32 by ignoring or reusing some identified stop points, and we measured method coverage for each app with 90 minutes time limit as our testing budget.

All experiments are conducted on a private cloud that has 24 virtual processor cores. A Google Nexus 4 smartphone serves as the remote desktop client.

5.2.3 Experimental Results

Figure 3 shows our scalability study results. For each app, we plot the the percentage method coverage progress against testing time. The findings of our experiment are listed as follows.

- In all cases, PUGA with 32 emulators achieved best test coverage than all of other scales within the given time budget. Compared with the single-threaded UGA, multiple-threaded PUGA outperforms as to the five apps. This results confirms that PUGA can indeed boost the effectiveness of UGA technique.
- For all apps, exponential growth of emulators means a faster rate at which PUGA achieves method coverage. For example, in *my effectiveness*, single-threaded UGA reached almost 40% method coverage when testing terminates. Whereas, using two emulators, PUGA reported the same coverage in 40 minutes, less than half of the time budget. When the number of emulators increases double sequentially, the time limit is nearly cut to only 15 minutes. For both of 16 and 32, in the first 10 minutes, PUGA has already surpassed 45%. This implies that ATT is capable of conducting parallel testing and PUGA is a far better choice when the testing time is limited.
- In the first 10 minutes, all the cases have a rapid growth of method coverage. This results from the fact that the recording of user traces happened during this period with relatively few redundant events compared the inefficient RND strategy we adopted.

Our quantitative experiment further validates that our intuition that PUGA is capable of leveraging human intelligence more quickly than the original UGA [16] for the identical user trace. Moreover, it confirms that parallel testing

We denote 1, 2, 4, 8, 16, 32 emulator(s) as 1×, 2×, 4×, 8×, 16×, 32×, respectively in the plotting.

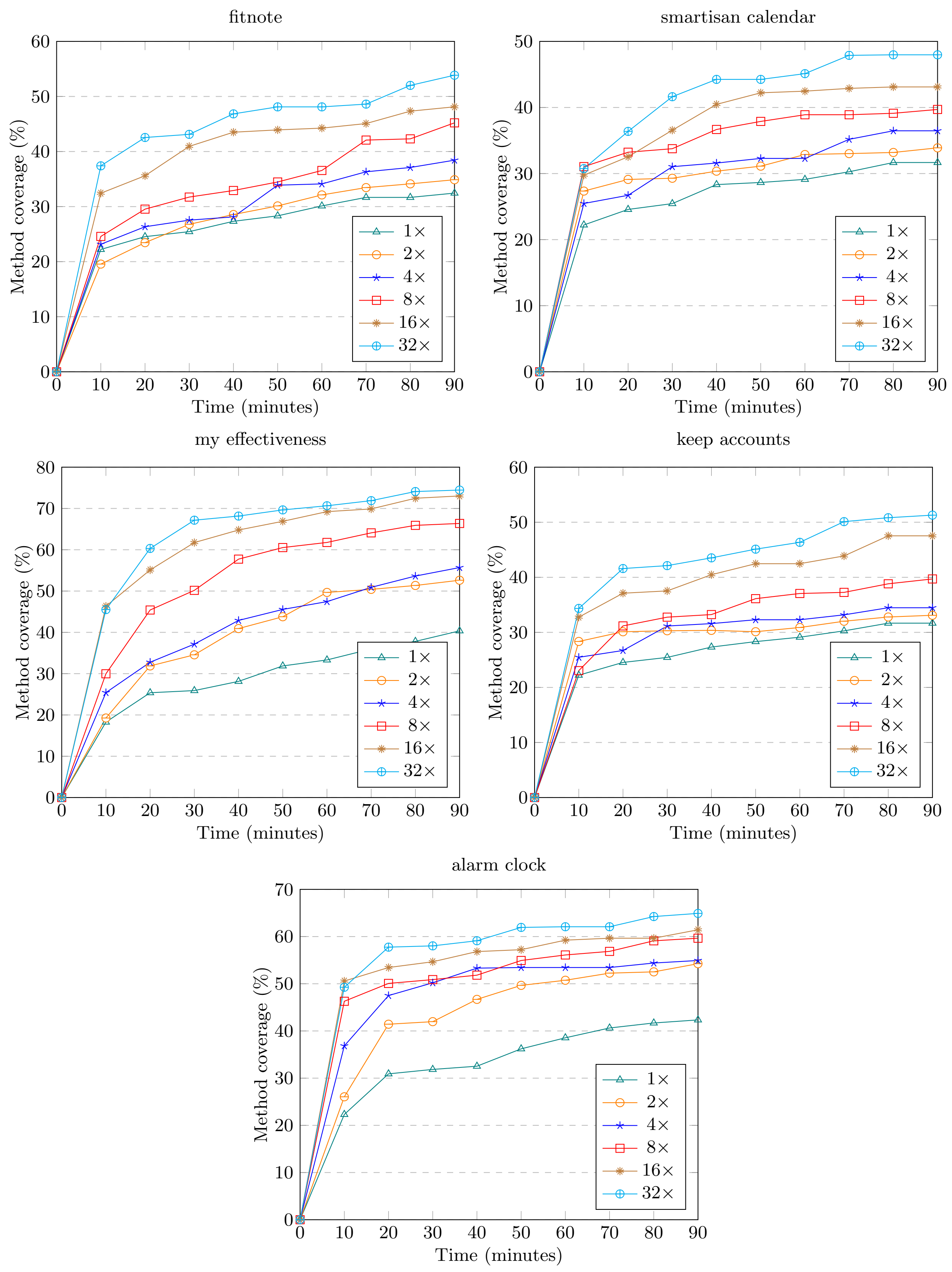

Figure 3: PUGA scalability study

Table 5: Subject apps

Name	*Category*	*#Instructions*	*#Activities*	*#Methods*
fitnote	fitness tracker	32683	11	4705
smartisan calendar	calendar	16752	31	1416
my effectiveness	to-do list and note	11305	39	2896
keep accounts	tally book	50840	55	2456
alarm clock	alarm clock	36861	17	741

of ATT scaled well, and can cover much more code than single-threaded UGA within the given time budget.

6. RELATED WORK

There are several existing Android testing tools for generating inputs of mobile apps. The primary goal of these tools is to aid developers detecting faults in apps, preventing them to be leaked into releases. Thus, to improve the effectiveness of such tools, the testing input generation techniques are supposed to generate relevant inputs to explore as much space of app under test as possible.

According to the interactive nature of mobile phones, Android apps are driven by asynchronous events, and input generation for such apps is equivalent to generating input event sequences (e.g., UI events like click/touch on the screen, and system events such as the notification of low battery level or received SMS). Testing tools can generate input events randomly or following various systematic strategies or heuristics. A testing tool can be while-box (guided by program source/bytecode) or black-box (purely guided by the app's user interface).

The baseline approach of event sequence generation is blindly generating events by random. Monkey [7], the most widely-adopted tool for testing Android apps, belongs to such category. Monkey is released with the Android SDK and can be easily used by pressing one button. Though being the most cheap way to test the app, Monkey is only capable of generating limited kinds of events, and is not aware of the app's structure. Its efficiency degenerates quickly when it is applied on apps that contain deep interactions within reasonable time budget, as Monkey generates too many redundant events to repeatedly exploring code that has already been covered. Realizing such limitations, Dynodroid [17] extends Monkey by adopting a heuristic random input generation scheme with the feedback from dynamic program analysis, achieving better effectiveness.

To more systematically explore the app, testing with GUI model is a promising direction. Such techniques use an app's GUI model to guide input event generation to avoid redundancy that is inevitable for random testing. A GUI model can either be built statically from the source code [25, 27], or be obtained dynamically at the program run [11, 28]. The prototype implementation of these model-based techniques showed significant improvements compared with random testing. For example, SwiftHand [11] is aimed at maximizing the coverage and minimizing the restarts of application under test. A^{3}e is capable of building the Static Activity Transition Graph of the app, which may work on the condition that some behaviors of the target app can only be revealed relied on specific inputs and systematic exploration strategies. EvoDroid [18] uses the evolutionary algorithm as the systematic exploration strategy to generate inputs that can transfer from one application state to another. These tools, however, only generates limited UI events (e.g., click or scrolling), and falls short for those codes that require specific actions to trigger.

GUITAR [22] is a testing automation framework for GUI-based applications. It is used for reverse engineering the applications, modeling apps as a graph and generating test cases which are used for the replaying process. GIUTAR has been applied to the range of Android apps and web apps. Robotium [24] is an Android testing automation framework that has full support for native and hybrid applications. Robotium makes it easy to write robust automatic black-box UI test cases for Android applications. With the support of Robotium, test tool developers can be devoted more to the generation of test scenarios, spanning multiple Android activities. However, both GUITAR and Robotium do not consider any implementation of system events generating and are not much convenient for developers to use. Recently, a few of research studies have notice the parallel testing of Android apps for the potential improvement of the testing efficiency. For example, GUITAR is released with an distributed extension to distribute and execute test cases on a cluster of slave testing nodes. However, GUITAR uses a centralized controller to generate all of the testing inputs, while our ATT is more flexible.

7. CONCLUSION

We proposed an Android testing framework ATT, which offers a set of APIs to facilitate testing tool construction. The APIs include: (1) UI, sensory and system event generation; (2) system control, profiling, monitoring and system state query; (3) program instrumentation and resigning. We implemented a Monkey+ tool to improve the behaviors of Monkey [7] and reproduced the UGA technique on ATT, which demonstrates that our framework helps a lot during the implementation and improvement of some testing tools. And we also implemented a parallel version of UGA called PUGA, and conducted experiments on a cloud platform. The results show that ATT is capable of conducting massive parallel testing on a cluster of emulators and devices. PUGA can cover much more code than single-threaded UGA [16] within the given time budget.

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) of China. It was also partially supported by Tencent, Inc. and by the Collaborative Innovation Center of Novel Software Technology and Industrialization of China.

8. REFERENCES

[1] Android Debug Bridge. `http://developer.android.com/tools/help/adb.html`.

[2] Android SDK. `https://developer.android.com/sdk/index.html`.

[3] Apktool Tool. `https://code.google.com/p/android-apktool`.

[4] F-Droid Free and Open Source App Market Website. `https://f-droid.org`.

[5] Google Play Website. `https://play.google.com/store`.

[6] Instrumentation. `http://developer.android.com/reference/android/app/Instrumentation.html`.

[7] Monkey Tool. `http://developer.android.com/tools/help/monkey.html`.

[8] 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*, page 59. ACM, 2012.

[9] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. *ACM SIGPLAN Notices*, 48(10):641–660, 2013.

[10] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a single model and test prioritization strategies for event-driven software. *Software Engineering, IEEE Transactions on*, 37(1):48–64, 2011.

[11] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, volume 48, pages 623–640. ACM, 2013.

[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*, pages 72–81, 2013.

[13] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83. ACM, 2011.

[14] C. Hu and I. Neamtiu. A GUI bug finding framework for Android applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1490–1491. ACM, 2011.

[15] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, page 18. ACM, 2014.

[16] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Lu. User guided automation for testing mobile apps. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, pages 27–34, 2014.

[17] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[18] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609. ACM, 2014.

[19] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripper: Reverse engineering of graphical user interfaces for testing. In *WCRE 2003, The 10th Working Conference on Reverse Engineering*, 2003.

[20] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 30–39. ACM, 2000.

[21] A. M. Memon and M. L. Soffa. Regression testing of GUIs. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127, 2003.

[22] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, 2014.

[23] A. Rauf, A. Jaffar, and A. A. Shahid. Fully automated GUI testing and coverage analysis using genetic algorithms. *International Journal of Innovative Computing, Information and Control (IJICIC) Vol*, 7, 2011.

[24] R. Reda and H. Josefson. Robotium, 2012.

[25] A. Rountev and D. Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 143. ACM, 2014.

[26] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A comprehensive security assessment. *IEEE Security & Privacy*, (2):35–44, 2010.

[27] W. Shin. Test model and test case generation based on static analysis for gui testing of android application. *Konkuk University, Ph. D. Dissertation*, 2013.

[28] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.

[29] X. Yuan, M. B. Cohen, and A. M. Memon. GUI interaction testing: Incorporating event context. *Software Engineering, IEEE Transactions on*, 37(4):559–574, 2011.