Replicable Testing of Android Applications with Replaykit

Dan Watson
daniel.watson.3141@gmail.com
University of Waterloo, SWAG

June 27, 2018

1 Setup

Note: In some cases, commands are too long to fit nicely into this document, so they have been split across a few lines. All such commands given in a single step are one single command and will only work as such.

1.1 Android Debugging Bridge

1. Make sure adb is working. Type:

   ```
   adb
   ```

   It should give you version information and instructions. If it does, the SDK and ADB are working, and you can go to step 4.

2. Install Android SDK cli tools.

3. Ensure <sdk>/platform-tools is in your PATH.

4. Check the ADB again to ensure everything is installed properly.

5. Enable USB debugging on the Android device.

6. Enable ”show touches” in the developer options as well. This will help us later.

7. Plug your phone into your computer with an MTP (Media Transfer Protocol) capable cable.

8. Ensure your phone has USB MTP enabled in developer options. You can also check if the phone is seen as a storage device by your OS. If so, your phone is in MTP mode.

9. Test the link by sending a command to the phone over the ADB. With the screen off, type:

   ```
   adb shell input keyevent KEYCODE_WAKEUP
   ```

   Your phone should activate.

1.2 Trepn Profiler

1. Install Trepn Profiler, available on Google Play Store.

2. We need to configure Trepn to record what we want to measure. Open trepn and go to settings>Data Points.

3. Select which data-points to collect. I recommend CPU Load(Normalized), GPU Load, Memory Usage, and Screen State. Screen state is to help with processing (more on that later). The others are relevant performance data-points.

4. Be sure to save your preferences. Open Settings>general>Save Preferences. Remember what you name this file, save it to a variable in your console, or write it into a text file so you can copy and paste it later.

5. We also need to locate the install directory for Trepn. Open your phone's file browser, or browse your phone from your computer and locate the "trepn" folder, usually in the root directory unless it was installed to an SD card. Get the path of this directory (relative to the phone, not your computer) and save it to a variable or write it down somewhere. Example: "root/trepn" or "sdcard/trepn".

1.3 ReplayKit

For more information, see: Replaykit's Github

1. Find the executable file called starfish.exe or appetizer.sh

2. From shell terminal in Portable/replaykit-master/<Your OS>, get the path to this executable and add it to your PATH.

3. For Mac and Linux, be sure to set the .sh file to executable. This is usually (but not always) unnecessary on Windows.

4. Test your installation. Type:

   ```
   cd..
   starfish #or
   appetizer
   ```

   depending on the name of the executable. It is starfish for Windows and appetizer for Linux and Mac.It should tell you "incorrect usage" as opposed to "command not found".

5. Now find your phone's uid, usually the same as the serial number. This is how we will identify the phone for our experiments. Type:

   ```
   starfish devices list
   ```

   You should see a list of attached Android devices. The first item in the list will be marked "uid". This is the field we need. Save it to a variable or copy it down (with quotes). Example:

   ```
   $myPhone = "abcd1234"
   ```

2 Basic Command-Line Functionality

This section is for reference and you need not type these commands into your terminal if you are just getting set up.

2.1 Android Debugging Bridge

We can send commands to the phone over the ADB. With your phone screen off, type:

```
adb shell input keyevent KEYCODE_HOME
```

Your phone should activate after a moment. If it does not, either the adb is mis-configured, or the phone is not connected properly. You can find lots of helpful adb commands HERE

2.2 Trepn Profiler

We can tell control trepn using the Trepn Profiler app.

1. You can load your preferences file in the settings menu. Using our $trepnPath and $prefsFile variables from before, we can also load our preferences in this way:

   ```
   adb shell am broadcast
   -a com.quicinc.trepn.load_preferences
   -e com.quicinc.trepn.load_preferences_file
   -join($trepnPath ,"/saved_preferences/", $prefsFile)
   ```

2. Start Profiling:

   ```
   adb shell am broadcast
   -a com.quicinc.trepn.start_profiling
   -e com.quicinc.trepn.database_file "trepnTemp"
   ```

3. Stop Profiling:

   ```
   adb shell am broadcast
   -a com.quicinc.trepn.stop_profiling
   ```

4. Convert output db file to csv

   ```
   adb shell am broadcast
   -a com.quicinc.trepn.export_to_csv
   -e com.quicinc.trepn.export_db_input_file "trepnTemp.db"
   -e com.quicinc.trepn.export_csv_output_file $outputFile
   ```

2.3 Replaykit

Any command we can send over the adb, we can send over starfish. For example:

```
starfish devices control $myPhone shell input keyevent KEYCODE_WAKEUP
```

2.3.1 Testing Multiple Devices

One thing Replaykit allows us to do easily that the adb does not is control multiple devices at once. This functionality is optional, but allows you to control multiple phones if you wish, and is fully compatible with the rest of the testing technique presented here.

1. We first create an array of uid's:

   ```
   $myPhones = $phone1,$phone2,$phone3
   ```

2. Then wake them all up:

   ```
   starfish devices control $myPhones
   shell input keyevent KEYCODE_WAKEUP
   ```

We can always use our array of phone id's instead of a single id in our scripts.

2.3.2 Recording touch interactions

For more information, see THE GITHUB.

1. Start recording:

   ```
   starfish trace record --device $myPhone mytrace.trace
   ```

2. Stop recording:

   ```
   exit
   ```

3. Check the trace file(optional):

   ```
   starfish trace info mytrace.trace
   ```

   This gives information about the trace such as duration and number of recorded events. If numbers seem off, something isn't working right. We can also view and edit the trace file by decompressing it, then opening it with a text editor. Replaykit is ok with compressed or decompressed trace files but they are compressed by default

   Play back the Trace:

   ```
   Starfish trace replay mytrace.trace $myPhone
   ```

   I recommend using CPU Load(Normalized), GPU Load, Memory Usage, and Screen State

   Screen state is to help with processing (more on that later). The others are relevant performance datapoints. Be sure to save your preferences by hitting general¿Save Preferences. Remember what you name this file (or save it to a variable in your console)

3 Designing and Running Experiments

1. Find the package name if the app you want to test:

   ```
   adb shell pm list packages
   ```

   This will show you all packages installed on the device. To narrow the list, type:

   ```
   adb shell pm list packages %filter%
   ```

   if you think you know any part of the package name.

   save the full package name as a variable:

   ```
   $myapp = "com.%me%.%myapp%"
   ```

2. Record a trace for our app by launching it, starting the trace, interacting a bit, then exiting the trace.

3. Now, to collect several samples we call our magic script that ties these things together:

   ```
   .\runtests $myPhone traces/mytrace.trace
   5 0 $myapp myProfile $trepnPath
   ```

   This will launch your target app and run your trace 5 times while profiling it with Trepn. Once it has completed its execution 5 times, it will save Trepn's output as a csv, which is then pulled into your working directory. Each Experiment(set of runs) generates 1 csv file which can then be processed in Excel, R, Matlab, or your spreadsheet software of choice.

3.1 Designing more Complex Experiments

Open runtests with your preferred text editor or IDE.

We can make our experiments more complex by adding other inputs like button presses and text.

```
starfish devices control $target
shell input keyevent KEYCODE_ENTER #--press enter

starfish devices control $target
shell input touchscreen text $city #--type something
```

More such actions can be found HERE.

To set up an experiment, first decide:

- What app shall I test?
- What series of interactions do I want to test?
- How consistent do I expect the app to be in how it responds?

Experimentation in this manner works ideally when: -UI elements that we wish to test are stationary on the screen (i.e. not moving) -The app is intended to give (largely) the same feedback to the same set of interactions (deterministic modulo user input) -Timing is not of critical importance to the outcomes of interactions.

Examples of Apps that fit this set of criteria: -Listing apps like reddit, airbnb, Netflix, Spotify, Tinder -Web browsers -Calculators -Puzzle Games such as Sudoku -Clock apps

Examples of Apps that do not work ideally: -Anything involving Maps like Google maps or Uber (as maps is somewhat non-deterministic as scroll speed is partly dependent on CPU constraints) -Any games involving motion or timing (flappybird, pong, etc.) -Any apps that update content continuously (such as Facebook, Instagram, Youtube) as consecutive sessions will not show the same content.

I do not mean to suggest we cannot use the tool to perform tests on these apps, but our sample variance is likely to be much higher and possibly more sensitive to small variations in experimental technique. Draw conclusions with caution when testing such apps.

Once you've chosen an app to profile, decide what interactions to test. Consider: -Does this set of interactions sufficiently resemble a typical use case? -Should this set of interactions consistently give us the same response?

Example:

Open propertyExp.ps1. In this file, we've scripted an experiment to test a very simple listview demonstration app, like the one found here: https://www.raywenderlich.com/178012/reac native-tutorial-building-android-apps-javascript Our script clears the text box, types in the name of a city (from a list of 3), presses enter, then scrolls the list up and down for a little while, before hitting the back button and returning to the main screen.

Suggestions:

As mentioned previously, certain games make good candidates for testing.

Web browsers make excellent candidates for this tool, as for most websites, the user sees the same content when they navigate to them. One could, for example, design an experiment to test the efficiency or quickness of web browsers by tapping the url bar, typing out a website, and pressing enter.