Kraken-Mobile: Cross-Device Interaction-based Testing of Android Apps

William Ravelo-Méndez, Camilo Escobar-Velásquez[1], Mario Linares-Vásquez[1]
Universidad de los Andes, Bogotá, Colombia
{wr.ravelo, ca.escobar2434, m.linaresv}@uniandes.edu.co

***Abstract*—Mobile applications that involve the interaction of two or more users are becoming more common nowadays, and the demand of good performance and availability by their users is increasing. This makes testing and automation of processes essential for delivering high-quality cross-device apps. In this paper, we present `Kraken`, a cross-device testing tool that enables a tester to write, run, and validate test scenarios that involve the interaction of more than two devices as well as its underlying implementation. The proposed tool uses APKs as input along with tests scripts wrote using the Gherkin syntax. Kraken is publicly available on GitHub (https://bit.ly/2Xjd4Gq). Videos: https://bit.ly/2x0qE2A**

***Index Terms*—Signaling; Automated Testing; Cross-device Testing; Cross-application Testing**

I. Introduction

Mobile applications are having an important role in the life style of human beings, providing users with support for simple and complex tasks. Mobile apps allow also for interaction in cross-device scenarios as in communication apps where two or more users interact with each other via messages and video calls. However, despite automated testing of mobile apps is a very prolific area, it has focused on generating inputs for single-user interaction scenarios [1], [2]; automated testing of apps that depend on cross-device interactions is not a well explored field yet.

Consider for example the scenario where a passenger requests a ride; then a driver accepts this request, travels for some time, and then finishes the route. Even though this may seem as a simple scenario, it requires the communication of two different apps (rider and driver app) running on different devices in order to complete the featured scenario. Cases like the aforementioned one, are becoming more common in mobile apps (*e.g.,* WhatsApp, Messenger, Uber, Lyft) [3]. Solutions for testing cross-device scenarios are available (*e.g.,* Octopus [4]), however, those tools are not publicly available, and have limitations in terms of the type of supported inputs.

With the purpose of overcoming the lack of publicly available tools for testing cross-device interaction apps, in this paper we present **`Kraken`**, an open source automated android E2E testing tool for scenarios that involve inter-communication between two or more users/devices. It works in a black box manner and uses a signaling-based protocol for coordinating the communication between devices. **`Kraken`** is publicly available on GitHub (https://bit.ly/2Xjd4Gq) and as a ruby gem with the package name `kraken-mobile`.

[1]Escobar-Velásquez and Linares-Vásquez were partially supported by a Google Latin American Research Award 2018.

II. Related Work

End-to-End testing (E2E) of mobile apps is widely supported by frameworks such as Calabash [5], UIAutomator [6], and Espresso [7]. For instance, Calabash [5] offers the possibility of testing in a black box manner while using Behavior-Driven Testing (BDT) and writing execution scenarios using the Gherkin sintas [8]. However, currently available frameworks for E2E testing of mobile apps do not support inter-communication of two or more users that interact with the app on different devices. Other approaches such as Sapienz [10] and MonkeyLab [11] provide automated inputs and test cases generation, however, there is also the cross-device interaction limitation.

A potential solution could be Appium [12], that creates a server for enabling communication with the app under test; therefore, multiple servers could be launched in order to test in parallel different apps or devices. However, Appium does not support communication between devices and requires an user to automate the provisioning of the different servers and configurations.

Given the limitations of existing mobile testing framewoks, the Uber Engineering Team created a solution named Octopus [4]. This framework allows devices to have a virtual two-way communication via a test host (*i.e.,* a computer running test cases and connected via USB to the devices) responsible of orchestrating signals. Despite Octopus proposing a solution to the inter-communication testing problem, it is not publicly available for users outside of Uber.

III. The **`Kraken`** Tool

To allow cross-device interaction-based testing, our **`Kraken`** tool is inspired on both E2E existing frameworks and Uber's octopus. We build on top of the specification-by-example capability provided by cucumber/calabash and combine it with the signaling approach proposed by Uber's octopus. **`Kraken`** includes five additional capabilities not supported by existing E2E testing frameworks: (i) send signal, (ii) read signal, (iii) generate random events over full- and part-of-screen, (iv) *kraken* monkey (*i.e.,* random signals and events), and (v) define a data pool of sensitive data such as passwords or API keys that should be used within the test cases. These features will be explained in Section IV-A.

Fig. 1 presents the architecture of **Kraken**. It consists of several modules aimed at launching required devices, creating artifacts for each device, supporting/orchestrating the signaling protocol, executing the steps and generating reports.

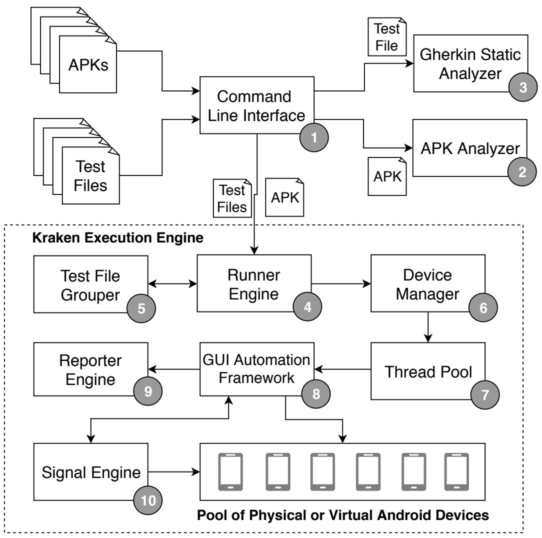


Fig. 1: The **Kraken** architecture and workflow.

A. *Signaling*

The signaling approach is based on the "publish-and-suscribe" architectural style, in which the actors (*i.e.,* devices) wait for signals indicating when to start the execution of a set of steps. To achieve this, **Kraken** has a central Signal Engine that listens for signals from devices and sends execution actions to the corresponding devices timely; it is worth noting that this process is not done serially, which allows for parallel execution of scenarios when required.

The *Signal Engine* coordinates each device and enables their communication by using a file-based signaling protocol; each device has a text file in its internal file system called inbox where it can receive signals from other devices. These signals are strings written to the inbox file that indicates an action completed in another device. The lack of a message allows a device to recognize an execution point where it should wait for an incoming signal. To accomplish this, **Kraken** offers two main functions: *readSignal* and *writeSignal*.

The *writeSignal* function sends a signal to another specified device, to achieve this, it receives the following parameters: (i) *channel*, a string containing the tag of the device that is going to receive the signal (*e.g.,* @user1); and (ii) *content*, a string that describes the action completed by the current device and that represents the signal name (*e.g.,* REQUESTED_RIDE).

Finally, to write the content of the signal into the desired inbox file, **Kraken** translates the channel parameter into a device id and uses the Unix *echo* command with the help of the Android Debug Bridge (ADB) to modify the inbox file.

The *readSignal* function waits for a signal to be written in the current device inbox; it uses the following parameters: (i) *channel*, a string containing the tag of the device that is going to receive the signal (*e.g.,* @user1); (ii) *content*, a string containing the signal name that is expected to be received (*e.g.,* REQUESTED_RIDE); and (iii) *timeout*, an integer representing the number of seconds the device is going to wait for a signal.

Finally, to wait for a signal in the desired inbox, **Kraken** translates the channel parameter into a device id and runs a loop that every second reads the last line of its inbox file and compares it to the expected content.

B. *Scenarios execution*

Using the *Command Line Interface* ① a user can upload a set of *.apk* files under test and the test files written using the Gherkin and **Kraken** syntaxes. Once the user has provided these files, (i) **Kraken** validates the provided APKs are signed with the **Kraken** user's keystore (using the *APK Analyzer* ②), and (ii) via the *Gherkin Static Analyzer* ③ component, it checks that the test files are in compliance with the Gherkin and **Kraken** syntaxes.

To start the testing process, the *Runner Engine* ④ checks the number of available devices or emulators and with the help of the *Test File Grouper* ⑤ component, **Kraken** maps each test file with a device where it is going to be executed. Next, the *Runner Engine* ④ notifies the *Device Manager* to start the testing process in the required devices or emulators, each one of them running on a different thread ensuring an isolated environment; also the *Device Manager* ⑥ will create an inbox file in each device where the signals are going to be received. Every test process starts thanks to the *GUI Automation Framework* ⑧ component in charge of interpreting every instruction specified in the test files, executing the instruction on a device, asserting that the action is completed correctly and finally saving a report.

To implement the signaling actions, when the *GUI Automation Framework* ⑧ reads one of these instructions in the test file it pauses its execution and notifies the *Signal Engine* ⑩. When it is required to send a signal from one device to another, the *Signal Engine* writes the signal content to the destination inbox file and informs the *GUI Automation Framework* ⑧ that the action has been completed in order to continue the processing of instructions.

On the other hand, when a device needs to wait for a signal, the *Signal Engine* ⑩ creates a loop that continuously reads the device inbox file and returns when the expected signal content is received or the default timeout is reached. Finally, when all instructions have been processed and completed, the *GUI Automation Framework* ⑧ notifies the *Reporter Engine* ⑨ component in order to read the logs and results saved, and generates an HTML report.

IV. Kraken in Action

A. *Console Interface*

User interaction with **Kraken** is done via console commands. Once the **Kraken** gem is installed, users can (i) generate a test folder skeleton, (ii) list the devices attached to the computer, (iii) setup the user tag for each device (*i.e.,* the way each device is identified in the **Kraken** framework), (iv) sign an apk with the currently configured keystore, and

(v) run a test. The first step to use **Kraken** is generating the base project that contains a *.feature* test file (similar to the one in Snippet 1), a subfolder containing a ruby file for the step_definitions, and a subfolder with support files required for **Kraken** execution; to do this, the user should call the `kraken-mobile gen` command.

Snippet 1: Example test file

```
1  Feature: Example feature
2
3  @user1
4  Scenario: As a first user I say hi to a second user
5  Given I wait
6  Then I send a signal to user 2 containing "hi"
7
8  @user2
9  Scenario: As a second user I wait for user 1 to say hi
10 Given I wait for a signal containing "hi"
11 Then I wait
```

A test file should contain scenarios definition following a Gherkin+**Kraken** syntax. For example, the feature in Snippet 1 contains two scenarios, one per each device involved in the test. However, as it can be seen in lines 3 and 8 of Snippet 1, each scenario is linked to a specific user (*i.e.,* a device). The user tag follows a naming pattern: `@user(\d+)`. The tags are automatically assigned by **Kraken** to each device connected to the execution engine; the tags can be checked a-priori by using the `kraken-mobile devices` command, which presents the devices information (including the tags). The following snippet is an example of the command output:

```
$ kraken-mobile devices
List of devices attached
user1 - emulator-5554 - Android_SDK_built_for_x86
user2 - emulator-5556 - Android_SDK_built_for_x86
```

If a user wants to modify the setup parameters for a test (*i.e.,* amount of devices, tag assigned to each device and apk to be used in each device), she can do it via the `kraken-mobile setup` command. Examples of the output can be seen in the following snippet:

```
$ kraken-mobile setup
How many devices do you want to use? 2
Choose your user 1 emulator-5554
What APK user 1 is going to run? ~/Desktop/app.apk
Choose your user 2 emulator-5556
What APK user 2 is going to run? ~/Desktop/app.apk
Saved your settings to .kraken_mobile_settings. You can edit the
        settings manually or run this setup script again
```

Due to the usage of calabash as part of the architecture, **Kraken** requires the APKs under test to be signed with the same keystore as the one currently configured; it is done by using the `kraken-mobile resign <apkPath>` command. Once the setup is done, the user must modify the *.feature* file provided by **Kraken**, with her specific requirements. It is worth remembering that **Kraken** uses the Gherkin + **Kraken** syntax, thus, the tester can use any of the steps already defined in Calabash [5]. The **Kraken** specific steps are:

1) Send signal: As it was mentioned in Section III-A, **Kraken** provides a step that sends a signal to a device. This step has two parameters: the user (*i.e.,* tag) that will receive the signal and the content of the signal. The structure of this step is:

```
I send a signal to user (\d+)containing "([^\"]*)"
```

2) Read signal: This step can be used in three different ways: first, to define only the timeout to receive a signal (See line 1 in Snippet 2); second, to set the expected content (see line 2 in Snippet 2); and third, to define the expected content and timeout (see line 3 in Snippet 2)

Snippet 2: **Kraken** send signal

```
1  I wait for any signal for (\d+) seconds
2  I wait for a signal containing "([^\"]*)"
3  I wait for a signal containing "([^\"]*)" for (\d+) seconds
```

3) Random Events Scenario: **Kraken** provides a scenario step that allows users to generate GUI-based random inputs by following the syntax presented in Snippet 3. This step can execute a given number of random events over the full screen (*line 1 in Snippet 3*) or on a specific region defined by the user as a portion of the screen (*line 2 in Snippet 3*).

Snippet 3: **Kraken** random step

```
1  I start a monkey with (\d+) events
2  I start a monkey with (\d+) events from height (\d+)% to (\d+)%
       and width (\d+)% to (\d+)%
```

4) Kraken Monkey: **Kraken** introduces another step that also sends and reads signals randomly, in addition to the rest of events (text input, tap, etc.). The structure of this step is:

```
I start a kraken monkey with (\d+)events
```

5) Properties file: **Kraken** uses properties files to store sensitive data such as passwords or API keys that should be used in your test cases. The properties files should be a manually created JSON file with the structure presented in the following snippet:

```
{
    "@user1": {
        "PASSWORD": "test"
    }
}
```

Note that the key-value pairs are organized by user tags. In order to use the values defined in the properties file, the property should be invoked using "< ... >" as in the following:

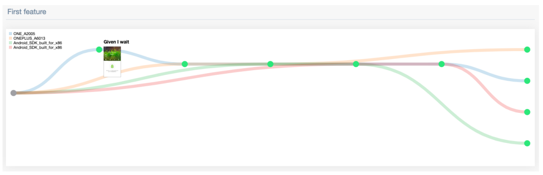
```
I see the text "<PASSWORD>".
```

Finally, after writing the scenarios, the last step for using **Kraken** is running it with the `kraken-mobile run <apk>` command; it must be called in the folder that contains the *.features* files. However, as it was mentioned before **Kraken** can be used with a different APK for each device. Therefore, once *setup* has been done, the command must include the flag: `--configuration=<conf_file_path>`. In case the user wants to use a property file, then the command must include `--properties=<properties_path>`.

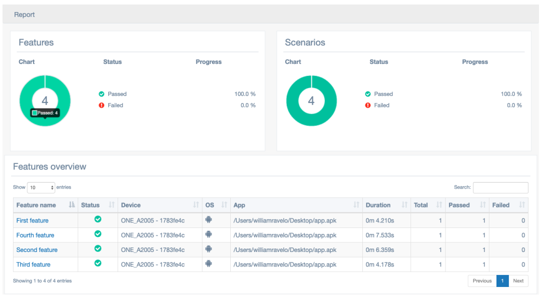
B. Web Reports

When finishing the execution of the test files, **Kraken** will generate a directory (in the current folder) that contains three reports: (i) general report, (ii) execution report by device, and (iii) execution detail by device.

General report: In the top section of this report **Kraken** provides metadata of all the devices used in the execution of the test files. A Sankey diagram is included with the purpose of showing the actions flow of each device in every feature, as in a timeline (Fig. 2a). In this type of graph, the nodes will represent an action performed by a device; when the viewer hovers over the node it shows the executed step action as well as a thumbnail showing the GUI state after the action was completed. If the color of the node is green it represents that the action passed, otherwise, if the node is red then it indicates an error in the execution. Each path, composed of edges with the same color, represents a different device.



(a) General report - sankey diagram



Report

Features

Chart | Status | Progress

4 — Passed: 4

✓ Passed 100.0 %

● Failed 0.0 %

Scenarios

Chart | Status | Progress

4

✓ Passed 100.0 %

● Failed 0.0 %

Features overview

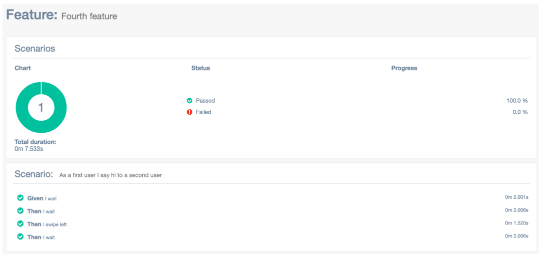
Show 10 entries Search:

Feature name	Status	Device	OS	App	Duration	Total	Passed	Failed
First feature	✓	ONE_A2005 - 1783fe4c	🤖	/Users/williamravelo/Desktop/app.apk	0m 4.210s	1	1	0
Fourth feature	✓	ONE_A2005 - 1783fe4c	🤖	/Users/williamravelo/Desktop/app.apk	0m 7.533s	1	1	0
Second feature	✓	ONE_A2005 - 1783fe4c	🤖	/Users/williamravelo/Desktop/app.apk	0m 6.359s	1	1	0
Third feature	✓	ONE_A2005 - 1783fe4c	🤖	/Users/williamravelo/Desktop/app.apk	0m 4.178s	1	1	0

Showing 1 to 4 of 4 entries

Previous 1 Next

(b) Features report by device



Feature: Fourth feature

Scenarios

Chart | Status | Progress

1

✓ Passed 100.0 %

● Failed 0.0 %

Total duration:
0m 7.533s

Scenario: As a first user I say hi to a second user

✓ **Given** I wait 0m 2.001s

✓ **Then** I wait 0m 2.006s

✓ **Then** I swipe left 0m 1.520s

✓ **Then** I wait 0m 2.006s

(c) Feature details by device

Fig. 2: Examples of reports generated by **`Kraken`**.

Execution report (by device): In this report, the tester can see the results of the tests, given a specific device. It includes the number of tests failed/passed, APK path, execution time, device name, and feature name (Fig. 2b).

Execution detail (by device): This final report presents the results of a specific test for a given device. The top section presents the number of steps passed and failed, in addition to the list of steps executed in either case. Finally, in the bottom section of the report, every step is presented with a screenshot of the device after the action was executed (Fig. 2c).

Source code, examples, more detailed explanations of the commands, and videos showing **`Kraken`** on action are available at https://bit.ly/2Xjd4Gq.

C. *Usage Examples*

To show **`Kraken`** capabilities we created and executed test scenarios for 10 different Android apps that involved the interaction of two or more users/applications. The capability of the framework to coordinate the signal protocol between two or more devices running the same application is illustrated with (i) social and messaging applications such as Tumblr, AskFM or F3 were one user shares information and then other users can access it; (ii) trivia games such as QuizUp or Spunky where two or more users compete by answering a list of questions; and (iii) transportation apps such as Picap were rider and driver interact.

Next, we used **`Kraken`** to test scenarios that involved the coordination of two or more devices running in different apps by testing a scenario where one user plays a song with an audio streaming service app such as Spotify, and then another user listens to the song and recognizes it with a song identifier app such as Shazam. Finally, we used **`Kraken`**'s capability to run random events over the GUI by testing multiplayer mobile games such as Tic Tac Toe, Infinite Words or Stick Men Fight, where a user can (i) invite another player to the game (ii) accept invitation from another player, (iii) wait for another player to make an action before making a move, and (iv) generate fast touch events over the GUI.

The videos showing the scenarios and the testing artifacts are available at https://bit.ly/2RgwDd8.

V. Conclusion & Future Work

We presented **`Kraken`**, the first publicly available tool for cross-device interaction-based testing of Android apps. **`Kraken`** allows developers/testers to write testing scenarios in an E2E fashion for Android apps that require interaction of two or more users on different devices. **`Kraken`** also allows developers/testers to combine E2E scenarios with random generation of events. Future work will be devoted to support the execution of scenarios in iOS devices and include a systematic exploration mode to enable cross-device GUI ripping.

References

[1] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *ICSME'17*, Sep. 2017, pp. 399–410.
[2] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *ASE'15*, 2015, pp. 429–440. [Online]. Available: https://doi.org/10.1109/ASE.2015.89
[3] S. Moore. (2015) Gartner says demand for enterprise mobile apps will out- strip available development capacity five to one. [Online]. Available: https://gtnr.it/2WvZRW8
[4] B. J. A. Chow. (2015) Octopus to the rescue: The fascinating world of inter-app communications at uber engineering. [Online]. Available: https://eng.uber.com/rescued-by-octopus/
[5] Calabash. (2019) Calabash-android. [Online]. Available: https://github.com/calabash
[6] Android. (2019) Ui automator. [Online]. Available: https://developer.android.com/training/testing/ui-automator
[7] ——. (2019) Espresso. [Online]. Available: https://developer.android.com/training/testing/espresso
[8] A. H. M. Wynne, *The Cucumber for Java book*, 1st ed. Pragmatic Bookshelf, 2012, pp. 4–5.
[9] Google, "Create ui tests with espresso test recorder. https://developer.android.com/studio/test/espresso-test-recorder."
[10] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. of ISSTA'16*, 2016.
[11] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *MSR´15*, ser. MSR '15, 2015.
[12] J. Foundation. Appium. [Online]. Available: http://appium.io/