Speeding up GUI Testing by On-Device Test Generation

Nataniel P. Borges Jr.
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
nataniel.borges@cispa.saarland

Jenny Rau
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
jenny.hotzkow@cispa.saarland

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.saarland

ABSTRACT

When generating GUI tests for Android apps, it typically is a separate test computer that generates interactions, which are then executed on an actual Android device. While this approach is efficient in the sense that apps and interactions execute quickly, the communication overhead between test computer and device slows down testing considerably. In this work, we present DD-2, a test generator for Android that tests other apps on the device using Android accessibility services. In our experiments, DD-2 has shown to be 3.2 times faster than its computer-device counterpart, while sharing the same source code.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Dynamic analysis*; • **Human-centered computing** → Graphical user interfaces; Smartphones.

KEYWORDS

dynamic analysis, test generation, Android

ACM Reference Format:
Nataniel P. Borges Jr., Jenny Rau, and Andreas Zeller. 2020. Speeding up GUI Testing by On-Device Test Generation. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3324884.3415302

1 INTRODUCTION

In recent years several tools and approaches have been proposed to automatically test Android apps. A recent survey [7] lists over 100 tools presented in major research conferences over the last nine years. The majority of these apps is developed following a *client-server* model, where a small app is deployed to the device (the client) and the main test generation logic remains on the computer (the server). This has the advantage of deferring complex computations, such as symbolic constraint solving to resolve path conditions, to a powerful machine. On the other hand, this setting involves a significant overhead in *communication* between client and server. To ensure that generated interactions are executed correctly, test generators have to *synchronize* with the user interface, waiting for a user element to actually show up before acting on it. For each interaction, the client first has to notify server if the UI is ready; the server then has to query the interface and/or the current state of the app; and then finally send the generated interaction to the device again.

Instead of synchronizing with the UI, a test generator can also simply *assume* that the UI is ready, relying on fixed action intervals; but the smaller the interval, the higher the risk of missing out UI elements; and the higher the interval, the slower the test execution. Ignoring the UI entirely and simply firing out random events, such as Android's default test generator Monkey, avoids synchronization and its associated problems entirely; but this also means it cannot perform complex operations or adapt the exploration based on visited UI elements.

In this work, we explore and introduce an efficient alternative. Rather than running the test generator on a separate server, we run it *directly on the device to be tested.* Not only does this make the communication between test generator and app under test much more efficient; it also drastically simplifies the test setup. Our *Device DM-2 (DD-2)* test generator is based on DM-2 [4]. It uses the Android *accessibility service* to access and interact with the app to be tested. Accessibility service is the mechanism provided by the Android framework to assist users with disabilities in using their devices. It enables the development of background services that are notified by the OS when events, such as focus change or when a UI transition occurs. Additionally, it provides mechanisms to determine the content of a screen, and interact with its elements.

Automated test generation and accessibility share several characteristics. From an accessibility perspective: to *use* an app, an *accessibility service* must identify the UI content and perform the interaction chosen by the *user*. Similarly, from an automated test generation perspective: to *test* an app, a *test generator* must identify the UI content and perform the interaction chosen by its *internal algorithm*. While test generators do not actively connect to the accessibility service, most rely, at some level, on Android's native UiAutomation[1], which itself is a wrapper around the accessibility service.

Being an app, DD-2 addresses several of the limitations mentioned above. It runs entirely on the device and, thus is highly scalable. One can, for example, trivially run the tool in parallel on several devices. Moreover, each successive version of Android provides compatibility for apps that were built using the APIs from previous platform versions[2], making it less likely that DD-2 must be adapted to run on newer Android versions. Being an accessibility service offers additional benefits. DD-2 does no require predefined action intervals; it instead listens for accessibility events triggered by the OS and acts when these events stop, indicating that the app

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3415302

[1] https://developer.android.com/topic/libraries/testing-support-library/index.html
[2] https://developer.android.com/guide/practices/compatibility#Versions

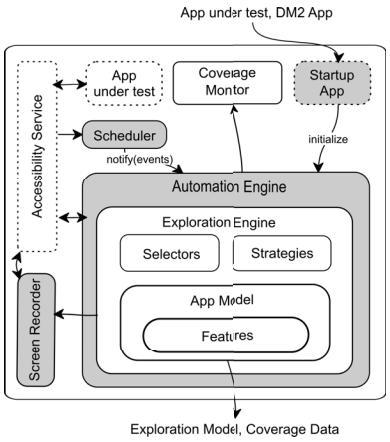

Figure 1: DD-2 architecture

is stable. Moreover, it has direct access to the accessibility events and can produce any event supported by Android accessibility service. Finally, DD-2 is open source and can be easily extended and adapted.

The remainder of this paper is organized as follows: we introduce DD-2's design and our experiments to evaluate its efficiency and effectiveness in Section 2 and 3. After discussing related work in Section 4, we show some usage scenarios for DD-2 (Section 5) and we conclude this work in Section 6.

2 TOOL DESIGN

DM-2 was developed with the goal of easy extensibility. Thus it is composed of different components that can be easily customized. Its Exploration Engine contains strategies that decide the next action based on the current screen and the previous exploration path. Its *selectors* decide which of these strategies to activate, and its App Model encodes all explored app screens and actions triggered along the exploration path. DD-2 reuses these components *as they are*, thus providing complete compatibility with previously developed extensions.

The main difference between DM-2 and DD-2 lies in its automation engine. In particular, DM-2 executes the exploration engine on a host PC and communicates with a remote device via ADB and TCP through its automation engine. This requires not only a constant connection to the device but also produces significant overhead for the execution time and limits the test generator capabilities. For example, tools that utilize Android's UiAutomator must rely on fixed delays to ensure that actions on the UI have sufficient time to take effect and update the app's UI. In contrast, DD-2 uses a different approach for the automation engine: it is implemented as an accessibility service. It listens to the accessibility events fired by the app under test (AuT) and uses this information to determine when the UI stabilized. DD-2's architecture is shown in Figure 1. The dashed blocks represent external elements, such as the AuT and Android's accessibility service. The white blocks represent the components used *as is* from DM-2, and the gray boxes represent the components developed in DD-2. DD-2 core components are: Startup App, Screen Recorder, Scheduler, and the Automation Engine.

Startup App is used to request the necessary permissions and to enable the user to choose an app to test. It is necessary because, for security reasons, Android does not allow a background service to directly record the screen without first requesting the user's permission. The Startup App then activates the Screen Recorder and forwards the user for the device settings, where the user must activate the service (Automation Engine). This behavior is also required for security reasons, as the user must actively start any Android accessibility services through the device settings. To use DD-2 on devices without user interaction, one needs to write once a short unit test or automation script, such as *DM2Launcher*[3].

Screen Recorder is the component responsible for providing the *automation engine* with the visual screen representation, for more accurate state identification. It works by creating a virtual display and requesting the OS to clone the main display's content into this new virtual one. It does so through the official media projection API[4], ensuring that it will work on future Android versions. A known limitation of all screen recording mechanisms on Android is the capture of secure screens, i.e., those which developers intentionally mark as protected from capture. Such behavior is, for example, commonly used on banking apps to ensure privacy.

Scheduler is responsible for determining when the AuT's UI is stable and can be interacted with. It connects to the accessibility service and queues all incoming accessibility events related to the AUT's UI content. It has two triggers to determine if the UI has stabilized. The first trigger is an interval without new events. Whenever a change happens in the AuT, it creates an accessibility event, to which the Scheduler listens. If the Scheduler does not receive any events within a specified period, it assumes the AuT UI to be unchanged and notifies the Automation Engine. The secondary trigger is an action timeout. Some apps and widgets regularly trigger accessibility events, and such situations render the first trigger ineffective. To prevent the exploration from continuously waiting for a UI to update, the Scheduler has a hard timeout trigger. If the app has not stabilized within a maximum timespan, the Scheduler will notify the Exploration Engine to interact with whichever widgets are on the screen. This timeout frequently occurs reached on situations such as countdowns, where time continuously changes, and map components, which continuously reload.

Automation Engine works as the bridge between the device and the test generation strategies. It has two roles: fetching the current UI state when notified by the Scheduler and translating high-level interactions from the Exploration Engine into device commands. To fetch a UI state, the Automation Engine queries the Screen Recorder for a screenshot and the accessibility service for the existing widgets. It then translates this data into a UI state using the App Model and forwards it to the Exploration Engine, which decides how to interact with the AuT. The Exploration Engine produces high-level interactions, such as "launch the app" or "terminate exploration", which the *automation engine* translates into sequences of low-level accessibility events it forwards to the accessibility service. For example, it translates the "launch the app" interaction into: show and clean list of recent apps, open the apps

[3]https://github.com/natanieljr/dm2-launcher.
[4]https://developer.android.com/reference/android/media/projection/MediaProjection

menu, search for the app, and click on the app node. Finally, upon receiving the "terminate exploration" interaction, the Automation Engine closes the *AuT*, deactivates the accessibility service, and stores the extracted app model–including all triggered actions, visited app states and screenshots–and the related code coverage data (if this feature was enabled).

3 EMPIRICAL EVALUATION

We conducted a set of experiments to evaluate DD-2's efficiency and effectiveness. In particular we aim to answer the following questions: **RQ1 (Efficiency):** *are tests executed on device faster than their server-client based counterparts?* **RQ2 (Effectiveness):** *how does on-device testing impact code coverage?*

For our experiments, we compared DD-2 against DM-2. We opted for this comparison as DD-2 uses the same exploration strategies and app model as DM-2 (it directly imports DM-2's source code). Thus, the differences observed in the experiments depend only on DD-2's changes. We did not compare DD-2 against other tools, as this has already been done for DM-2 [3, 4, 8] and there are no changes in the exploration strategy. We executed our experiments on a set of 13 apps, randomly chosen from recent researches that used DM-2 [3, 4]. All executions happened on Android emulators, running Android 9 (API 28), and configured with 4GB of RAM, 4 CPU cores, and a resolution of 1440x2880 pixels, similar specifications to modern phones. We opted to run our evaluation on emulators as are more scalable than physical devices. Nevertheless, DD-2 runs normally on devices running Android 6 or newer.

3.1 Efficiency

This experiment's goal is to determine if on-device testing is faster than its computer-device (client-server) counterpart. This evaluation is necessary as Android devices have less computational resources (CPU and RAM memory) than a computer and it will be executing both the AuT and the test generator.

We configured DD-2 and DM-2 to perform 1000 actions on each app, including clicks, text inserts, swipes, and restarting the app. We repeated this experiment, with different random seeds, ten times for each app to mitigate noise, and we measured how much time it took for both tools to execute 1000 actions.

Our results are summarized in Figure 2. On average, DD-2 took 1092 seconds (18 minutes) to execute 1000 actions, while DM-2 needed 3557 seconds (59 minutes). These numbers show that, on average, DD-2 is 3.2 times faster than DM-2 while using the same exploration strategy and app model, indicating that fully on-device approaches can speedup testing considerably.

To determine the rationale behind this difference, we inspected the fastest and slowest explorations. DD-2's fastest app was *Munch*, a minimalistic RSS Reader, with 808 seconds, and its slowest app was *Fake Traveler*, which mocks the device's location, with 2308 seconds. On these same apps, DM-2 took, respectively, 2796 and 6566 seconds. The difference between DD-2 and DM-2 on both apps highlights the benefits of DD-2's UI stabilization mechanism. By relying on accessibility events instead of fixed delays, DD-2 can react faster to app changes, while DM-2 happened to restart the app several times. Moreover, the difference between times for *Munch* and *Fake Traveler* also highlight one of DD-2's limitations. Some

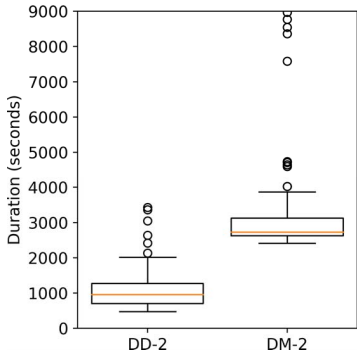

Figure 2: Duration of 1000 action run on DD-2 and DM-2

Android widgets, such as the map container, have frequent changes that are notified to the accessibility service, making the overall stabilization mechanism slower when they are present.

DM-2's fastest and slowest apps were *FOSDEM*, a companion app for the FOSDEM convention, and *Google Books viewer*, an app to search and view e-books, with 2540 and 7976 seconds respectively. This significant time difference also comes from the UI stabilization approach. *Google Books viewer* takes significantly longer to load data. Therefore, many times DM-2 attempted to interact with it while its UI was empty, assumed the app was stuck, and restarted it. DD-2 explored these same apps in 881 and 1499 seconds, respectively, further highlighting the benefits of its UI stabilization approach.

> *Fully on-device tests are on average 3.2 times faster than their computer-device counterparts.*

3.2 Effectiveness

This experiment's goal is to determine if the faster tests produced by our on-device test generator impact the overall test quality. This evaluation is necessary because of DD-2's UI stabilization mechanism. If the mechanism is fast but does not correctly wait for the UI to stabilize, it will not be able to trigger app functionality. As an evaluation metric, we used statement coverage, which has been extensively used to determine the effectiveness of testing tools and is regarded as a good predictor for fault detection [6].

We instrumented each app so that DD-2 and DM-2 could monitor statement coverage, and configured both tools to explore each app for 3000 seconds (50$minutes$). We repeated this experiment, with different seeds, ten times for each app to mitigate noise.

Our results are summarized in Figure 3. At 100 seconds, DD-2 achieves 9% more code coverage than DM-2, an advantage that falls to 6% at 500 seconds, and to 2.5% at 2000 seconds. While the results virtually converge in the end, this is expected as both DD-2 and DM-2 use the same exploration strategy and app model, which can only trigger a limited amount of app functionality as it is unlikely to reach complex test scenarios. Nevertheless, DD-2 reaches 99% of its maximum code coverage in 1052 seconds, while DM-2 takes 2372 seconds to do the same. Additionally, at 500 seconds, DD-2 reaches the same code coverage that DM-2 reaches in 2000 seconds. Both numbers indicate that DD-2 obtains coverage 2.2 times faster than

DM-2, thus indicating that faster exploration times of on-device testing lead to a faster increase in test coverage.

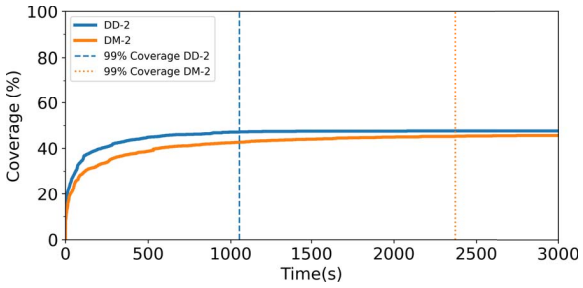

Figure 3: Coverage over time for DD-2 and DM-2

On average, fully on-device tests reached 99% of their coverage 2.2 times faster than their computer-device counterparts.

4 RELATED WORK

Automated test generation in mobile apps is an active research field with, according to a 2018 survey [7], more than 100 publications in the past nine years. Test input generation techniques are commonly classified into three categories [5]: *random*, *model-based*, and *exploratory*, according to their strategy to interact with the app.

Random testing tools interact with apps by creating random event sequences. Android's default test generator Monkey[5] and Dynodroid [9] are examples of tools in this category, with the former creating tests without UI synchronization and the later fetching the app UI to determine the next action. Tools that employ *model-based* strategies extract and use a model of the AuT to systematically generate inputs. The model can be extracted statically, such as in A^3E Targeted [2] or dynamically, as in AndroidRipper [1] and DM-2 [4]. Finally, tools which implement *exploratory* strategies employ a wide variety of techniques to guide test generation towards specific targets or increasing code coverage. Tools in this category include IntelliDroid [11], which uses concolic testing and Sapienz [10], which uses evolutionary algorithms.

DD-2 is built on top of DM-2 and reuses its exploration strategies; therefore, it falls into the *model-based* category. Nevertheless, by being built on top of DM-2, it also inherits its options to easily develop and integrate new exploration strategies and custom model abstractions.

5 USAGE SCENARIOS

We envision DD-2 as a tool beneficial to both research and industry due to its extensibility and scalability. Below we describe a few scenarios where DD-2 can be applied.

Scalable Testing: It can be deployed to multiple distinct devices to test if an app works under different configurations. Since DD-2 does not require a PC it can be easily used with online device farms. Moreover, DD-2 reuses DM-2 architecture and, thus, can be easily extended with new testing strategies. One can, thus, easily create strategies that guide testing on multiple apps simultaneously to explore multiple paths within the app in parallel.

Compatibility Testing: It can run on multiple device versions without change. Moreover, it inherits DM-2's record and replay capabilities. Therefore, DD-2 can be used to test how an app performs on multiple Android versions.

Continuous Integration: Testing is part of a regular development process. DD-2 provides fast tests and, thus, can be used as part of the continuous integration.

6 CONCLUSION

We present DD-2, an app that exploits the Android accessibility service to test other apps. DD-2 runs entirely on the device for increased performance, high scalability, and easy deployment. Since each successive Android version provides compatibility for apps built with APIs from previous versions, DD-2 should always be compatible with future versions of Android. Moreover, being itself an accessibility service, DD-2 is notified of each UI change and capabilities by the OS, which allows it to more effectively wait while the app UI stabilizes, improving test performance. Our evaluation showed that DD-2 is, on average, 3.2 times faster than the original DM-2, while reusing DM-2's exploration strategies and app model components out-of-the-box. This faster testing translates into faster code coverage, with our on-device approach taking 500 seconds to trigger the same amount of functionality that the original computer-device approach took 2000 seconds. DD-2 is freely available at https://github.com/uds-se/on-device-dm2.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.

[2] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of Android apps. In *Acm Sigplan Notices*, Vol. 48. ACM, 641–660.

[3] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. ACM, 133–143.

[4] Nataniel P Borges Jr, Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: a platform for Android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 916–919.

[5] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet? (E). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.

[6] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 72–82.

[7] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* 99 (2018), 1–22.

[8] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

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

[10] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.

[11] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *NDSS*, Vol. 16. 21–24.

[5] https://developer.android.com/studio/test/monkey