TimeMachine: Time-travel Testing of Android Apps

Zhen Dong
National University of Singapore
zhen.dong@comp.nus.edu.sg

Marcel Böhme
Monash University, Australia
marcel.boehme@monash.edu

Lucia Cojocaru
Politehnica University of Bucharest
lucia.cojocaru@stud.acs.upb.ro

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT

We introduce a prototype of an automated Android app testing technique. The distinctive feature of the technique is that it saves program states in the execution and restores an interesting state previously saved for further exploration when progress is slow so as to continuously explore new program behavior (namely time-travel testing). Our empirical results show that our technique outperforms the-state-of-art techniques including Sapienz (used at Facebook). Our technique has been developed as an fully automated Android testing tool and packed into an Docker image for ease of use. Our prototype is publicly available at https://github.com/DroidTest/TimeMachine.

1 TIME-TRAVEL TESTING

Idea. Time-travel testing is a mobile app testing approach, which saves app states in the execution and restores an interesting state previously-saved for further exploration when progress is slow in the hope of exploring new program behavior efficiently. An interesting state is identified based on runtime observations, e.g., a state is interesting or has potential to trigger new program behavior if the state is visited for the first time. App state is identified by GUI layout, two screens have an identifical state (abstractly) if they have the same GUI layout. App states can be saved by taking system snapshots when running on an emulator.

Boosting Mobile App Testing. Time-travel testing records interesting states observed, and can travel back to any previous state to launch a new execution. This enhances existing mobile app testing techniques to explore program behavior more efficiently. For instance, *random testing* (like Android Monkey) generates a very long sequence of events to exercise apps, but often suffers from low efficiency that execution loops in certain states (e.g., main screen). Time-travel testing allows random testing to jump out of such state loops and drive execution to infrequently visited states.

Search-based testing techniques systematically evolve a population of event sequences so as to achieve certain objectives such as maximal code coverage. The hope is that the mutation of fit event sequences leads to the generation of even fitter sequences. However, the evolution of event sequences may be ineffective. Pertinent app states which contributed to the original sequence's fitness may not be reached by a mutated event sequence because the original path through the state space is truncated at the point of mutation. Time-travel testing allows search-based techniques to identify the part of input events that leads to pertinent app states and avoid making any changes to this part during mutation, and generate inputs that explore the pertinent app states.

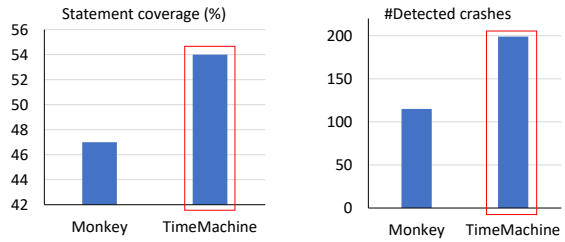

Figure 1: Statement coverage and number of crashes achieved by Android Monkey and TimeMachine.

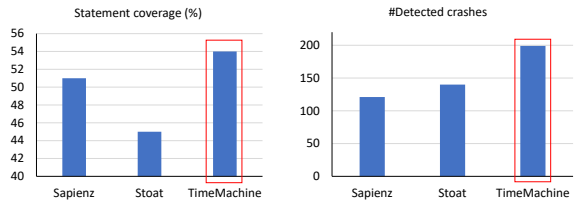

Figure 2: Statement coverage and number of crashes achieved by Sapienz, Stoat and TimeMachine on the open source benchmark.

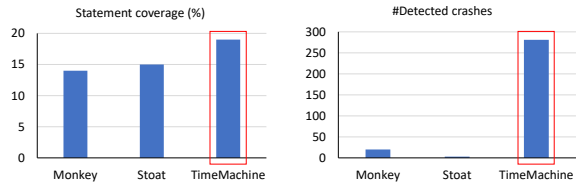

Figure 3: Statement coverage and number of crashes achieved by Monkey, Stoat and TimeMachine on the closed source benchmark.

Empirical Results. We develop the first time-travel-enabled test generator *TimeMachine* for Android apps by enhancing Android Monkey (the random testing tool released by Google) with our time-travel technique, and evaluated TimeMachine on three widely-used data sets. Experimental results show:

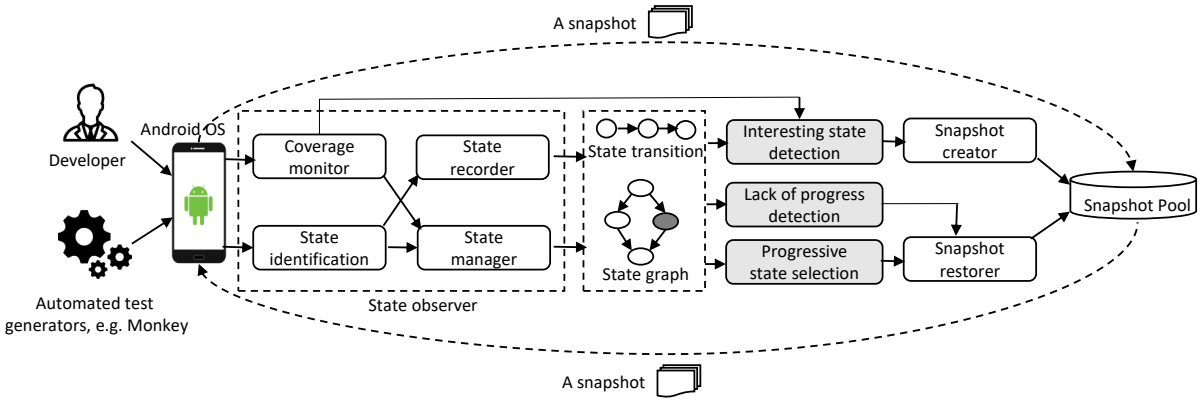

Figure 4: Time travel framework. Modules in grey are configurable, allowing users to adjust strategy according to scenarios.

TimeMachine achieves a significant improvement over Android Monkey in terms of statement coverage and the number of detected crashes. Meanwhile, TimeMachine achieves the best performance compared to the-state-of-art techniques on both open-source and closed-source benchmarks.

Figure 1 shows results achieved by Android Monkey and TimeMachine on the *AndroTest* benchmark [10]. AndroTest is a standard testing benchmark for Android testing containing 68 open source android apps and has been used to evaluate a large number of Android testing tools.

TimeMachine achieves 54% statement coverage on average and detects 199 unique crashes for 68 benchmark apps. Android Monkey achieves 47% statement coverage on average and detects 115 unique crashes. TimeMachine covers 1.15 times statements and reveals 1.73 times crashes more than Android Monkey.

Figure 2 shows results achieved by Sapienz, Stoat, TimeMachine on the AndroTest benchmark. Sapienz [12] is search-based Android app testing tool and used at Facebook. Stoat [13] is one of the most recent techniques for Android testing. These testing tools have also been adequately tested and are standard baselines in the Android testing literature.

TimeMachine achieves the highest statement coverage on average (54%) and is followed by Sapienz (51%), Stoat (45%). TimeMachine detects the most crashes (199) as well, followed by Stoat (140), Sapienz (121) and Monkey (48).

Figure 3 show results achieved by Monkey, Stoat, and TimeMachine on the *industial* benchmark [14]. IndustrialApps was a benchmark suite created in 2018 to evaluate the effectiveness of Android testing tools on real-world apps. The authors sampled 68 apps from top-recommended apps in each category on Google Play, and successfully instrumented 41 apps with a modified version of Ella [4]. In our experiment, we chose to use the original version of Ella and successfully instrumented 37 apps in Industrial app-suite. On this benchmark, we could not compare with Sapienz because the publicly available version of Sapienz is limited to an older version of Android (API 19).

TimeMachine achieves the highest method coverage 19% and the most found crashes 281 among all evaluated techniques. Compared to baseline MS and MR, TimeMachine improves method coverage to 19% from 17% and 15%, respectively.

Publication. Our work [11] was accepted on the 42nd international conference on Software Engineering (ICSE'20) (won the *ACM SIGSOFT Distinguished Paper Award*).

2 PROTOTYPE DESIGN & IMPLEMENTATION

We design a general time-travel framework for Android testing, which allows us to save a particular discovered state on the fly and restore it when needed. Figure 4 shows the time-travel infrastructure. The Android app can be launched either by a human developer or an automated test generator. When the app is interacted with, the state observer module records state transitions and monitors the change of code coverage. States satisfying a predefined criteria are marked as *interesting*, and are saved by taking a snapshot of the entire simulated Android device. Meanwhile the framework observes the app execution to identify when there is a *lack of progress*, that is, when the testing tool is unable to discover any new program behavior over the course of a large number of state transitions. When a "lack of progress" is detected, the framework terminates the current execution, selects, and restores the most progressive one among previously recorded states. A more *progressive state* is one that allows us to discover more states quickly. When we travel back to the progressive state, an alternative event sequence is launched to quickly discover new program behaviors.

The framework is designed as easy-to-use and highly-configurable. Existing testing techniques can be deployed on the framework by implementing the following strategies:

- Specifying criteria which constitute an "interesting" state, e.g., increases code coverage. Only those states will be saved.
- Specifying criteria which constitute "lack of progress", e.g., when testing techniques traverse the same sequence of states in a loop.
- Providing an algorithm to select the most progressive state for time-travelling when a lack of progress is detected.

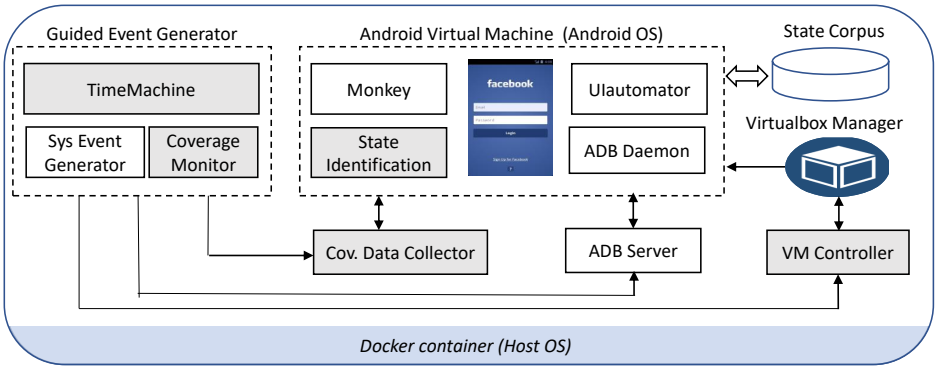

Figure 5: Architecture of TimeMachine implementation.

2.1 Taking Control of State

State identification. In order to identify what constitutes a state, our framework computes an abstraction of the current program state. A program state in Android app is abstracted as an app page which is represented as a widget hierarchy tree (non-leaf nodes indicate layout widgets and leaf nodes denote executable or displaying widgets such as buttons and text-views). A state is uniquely identified by computing a hash over its widget hierarchy tree. In other words, when a page's structure changes, a new state is generated.

To mitigate the state space explosion problem, we abstract away values of text-boxes when computing the hash over a widget hierarchy tree. By the above definition, a state comprises of all widgets (and their attributes) in an app page. Any difference in those widgets or attribute values leads to a different state. Some attributes such as text-box values may have huge or infinite number of possible values that can be generated during testing, which causes a state space explosion issue. To find a balance between accurate expressiveness of a state and state space explosion, we ignore text-box values for state identification. Our practice that a GUI state is defined without considering text-box values is adopted by previous Android testing works as well [8, 9].

State saving & restoring. We leverage virtualization to save and restore a state. Our framework works on top of a virtual machine where Android apps can be tested. A *virtual machine* (VM) is a software that runs a full simulation of a physical machine, including the operating system and the application itself. For instance, a VM with an Android image allows us to run Android apps on a desktop machine where related hardware such as the GPS module can be simulated. App states can be saved and restored with VM.

Our framework records a program state by snapshotting the entire virtual machine state including software and emulated hardware inside. States of the involved files, databases, third-party libraries, and sensors on the virtual device are kept in the snapshot so that the state can be fully resumed by restoring the snapshot. This overcomes the challenge that a state may not be reached from the initial state by replaying the recorded event sequence due to state change of background services.

2.2 Collecting State-Level Feedback

To identify whether a state is "interesting", our framework monitors the change in code coverage. Whenever a new state is generated, code coverage is re-computed to identify whether the state has potential to cover new code via the execution of enabled events. Our framework supports both open-source and close-source apps. For open-source apps, we collect statement coverage using the Emma coverage tool [5]. For closed-source, industrial apps, we collect method coverage using the Ella coverage tool [4]. For closed-source apps, statement coverage is difficult to obtain.

Our framework uses a directed graph to represent state transitions, where a node indicates a discovered state and an edge represents a state transition. Each node maintains some information about the state: whether there is a snapshot (only states with snapshots can be restored), how often it has been visited, how often it has been restored, and so on. This information can be provided to testing tools or human testers to evaluate how well a state has been tested and to guide execution.

2.3 Implementation

Our time travel framework is implemented as a fully automated app testing platform, which uses or extends the following tools: VirtualBox [1], the Python library pyvbox [7] for running and controlling the Android-x86 OS [3], Android UI Automator [6] for observing state transitions, and Android Debug Bridge (ADB) [2] for interacting with the app under test. Figure 5 gives an architectural overview of our platform. Components in grey are implemented by us while others are existing tools that we used or modified.

For coverage collection, our framework instruments open-source apps using Emma [5] (statement coverage) and closed-source apps using Ella [4] (method coverage). Ella uses a client-server model sending coverage data from the Android OS to the VM host via a socket connection. Unfortunately, this connection is broken every time a snapshot is restored. To solve this issue, we modified Ella to save coverage data on the Android OS to actively pull as needed.

On top of the time travel framework, we implement TimeMachine. TimeMachine works with the most widely-used version, Android Nougat with API 25 (Android 7.1). To collect state-level feedback, we modified Android Monkey and UI Automator to monitor state transition after each event execution. TimeMachine also

includes a system-level event generator taken from Stoat [13] to support system events such as phone calls and SMSs.

More importantly, we deploy the whole system into a docker container such that users can conveniently launch multple containers for parallel exectuions.

2.4 Artifact

Our time-travel Android app testing tool TimeMachine has been evaluated as Available and Reusable by ROSE (Recognizing and Rewarding Open Science in Software Engineering). We also made TimeMachine publicly available on the Github. Since released in Feb. 2020, the tool has been stared 67 times and been downloaded hundreds times by different universities or institutions. Our tool is at https://github.com/DroidTest/TimeMachine

REFERENCES

[1] 2018. VMWare VirtualBox. https://www.virtualbox.org/
[2] 2019. Android Debug Bridge. https://developer.android.com/studio/command-line/adb
[3] 2019. Android-x86. http://www.android-x86.org/
[4] 2019. ELLA: A Tool for Binary Instrumentation of Android Apps. https://github.com/saswatanand/ella
[5] 2019. EMMA: a free Java code coverage tool. http://emma.sourceforge.net/
[6] 2019. Google UI Automator. https://developer.android.com/training/testing/ui-automator
[7] 2019. A python library for VirtualBox. https://pypi.org/project/pyvbox/
[8] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 623–640. https://doi.org/10.1145/2509136.2509552
[9] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. 2018. DetReduce: Minimizing Android GUI Test Suites for Regression Testing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 445–455. https://doi.org/10.1145/3180155.3180173
[10] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. 429–440. https://doi.org/10.1109/ASE.2015.89
[11] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. 1–12.
[12] 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* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 94–105.
[13] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.
[14] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 738–748. https://doi.org/10.1145/3238147.3240465