

Time-travel Testing of Android Apps

Zhen Dong

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

Lucia Cojocaru

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

Marcel Böhme

Monash University, Australia
marcel.boehme@monash.edu

Abhik Roychoudhury

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

ABSTRACT

Android testing tools generate sequences of input events to exercise the state space of the app-under-test. Existing search-based 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. Our key insight is that pertinent *app states* which contributed to the original sequence’s fitness may not be reached by a mutated event sequence. The original path through the state space is truncated at the point of mutation.

In this paper, we propose instead to evolve a population of states which can be captured upon discovery and resumed when needed. The hope is that generating events on a fit program state leads to the transition to even fitter states. For instance, we can quickly deprioritize testing the main screen state which is visited by most event sequences, and instead focus our limited resources on testing more interesting states that are otherwise difficult to reach.

We call our approach *time-travel testing* because of this ability to travel back to any state that has been observed in the past. We implemented time-travel testing into TimeMachine, a time-travel enabled version of the successful, automated Android testing tool Monkey. In our experiments on a large number of open- and closed source Android apps, TimeMachine outperforms the state-of-the-art search-based/model-based Android testing tools Sapienz and Stoa, both in terms of coverage achieved and crashes found.

1 INTRODUCTION

Android app testing has been gaining in importance. In 2020, there is a smart phone for every third person (2.9 billion users) while app revenue will double from 2016 (US\$ 88 to 189 billion).¹ The number of bugs and vulnerabilities in mobile apps are growing. In 2016, 24.7% of mobile apps contained at least one high-risk security flaw [1]. The Android testing market is also expected to double in five years from US\$ 3.21 billion in 2016 to US\$ 6.35 billion in 2021.²

¹<https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/>
²<https://www.businesswire.com/news/home/20170217005501/en/>

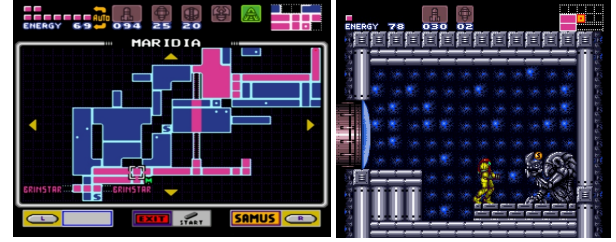
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7121-6/20/05...\$15.00

<https://doi.org/10.1145/3377811.3380402>



(a) Excerpt of the map of Maridia where pink marks explored rooms. (b) Samus discovering the Spazer weapon

Figure 1: Super Metroid (1994) on an Android Emulator

To illustrate the challenges of existing Android testing tools, take for example Super Metroid (Fig. 1), one of the best games for the NES gaming console, now available for Android. Super Metroid is played on a large map of rooms that can be explored in any order. By pushing the right buttons on the controller, the main character Samus moves from one room to the next, finding secrets and gaining in strength by fighting enemies. Today, Android app testing is like playing a game of Super Metroid, albeit *without* the ability to save after important milestones and to travel back in time when facing the consequences of a wrong decision.

One possible approach is to *generate a single, very long sequence of events* in a random fashion [3]. However, the testing tool may ultimately get stuck in dead ends. For instance, Samus may fall into pits or get lost in a particularly complex part of the labyrinth. This problem is overcome *only partially* by restarting the Android app because (i) we must start from the beginning, (ii) there is no clean slate, e.g., database entries remain, and (iii) how to detect when we are stuck is still an open question. For Android testing, the ability to save and travel back to the most interesting states goes a long way towards a more systematic exploration of the state space.

Another Android app testing approach [36] is to *evolve a population of event sequences* in a search-based manner. In each iteration, the fittest event sequences are chosen for mutation to generate the next generation of event sequences. An event sequence is mutated by adding, modifying, or removing arbitrary events. However, this approach does not allow for systematic state space exploration by traversing the various enabled events from a state. If e_i in the sequence $E = \langle e_1, \dots, e_i, \dots, e_n \rangle$ is mutated, then the suffix starting in e_{i+1} may no longer be enabled. For instance, when Samus stands next to an enemy or a ledge after event e_{i-1} and the event e_i is turned from a press of the \leftarrow -button to a press of the \rightarrow -button, Samus may be killed or get stuck. The remaining events starting from e_{i+1} become immaterial; rooms that were reached by E may not be reached by its mutant offspring.

In this paper, we propose instead to *evolve a population of states* which can be captured upon discovery and resumed when needed. By capturing and resuming an app’s states, we seek to achieve a systematic state space exploration (without going to the extent of exhaustive exploration as in formal verification). Due to the ability to travel back to any past state, we call this as time-travel testing. Our novel *time-travel testing* approach systematically resets the entire system—the Android app and all of its environment—to the most progressive states that were observed in the past. A *progressive state* is one which allows us to discover new states when different input events are executed. Once the tool gets stuck, it goes back in time and resumes a progressive state to execute different events.

We implement time-travel testing for Android apps into TimeMachine³ a time-travel-enabled variant of the automated Android testing tool Monkey [3]. In our example, one can think of TimeMachine as an automatic player that explores the map of Super Metroid through very fast random actions, automatically saves after important milestones, and once it gets stuck or dies, it travels back to secret passages and less visited rooms seen before in order to maximize the coverage of the map. Compared to tools that evolve event sequences, such as Sapienz [36], TimeMachine does not mutate the *sequence prefix* which is required to reach the fittest, most progressive state, and instead generates only the *sequence suffix* starting from that state. Compared to tools that generate a single, very long event sequence, such as Monkey [3] or Stoa [40], TimeMachine automatically detects when it gets stuck (i.e., there is a lack of progress) and resumes that state for further testing which is most promising for finding errors. In our experiments with Sapienz, Stoa, and Monkey on both open-source and closed-source Android apps TimeMachine substantially outperformed the state-of-the-art in terms of both, coverage achieved and errors found.

TimeMachine can be seeded with a set of initial event sequences. At the beginning of a testing session, TimeMachine takes a snapshot of the starting state. During test execution, TimeMachine takes a snapshot of every interesting state and adds it to the *state corpus*, travels back to the interesting state and executes the next test. For each transition from one state to another, TimeMachine also records the shortest event sequence. If no initial test set is provided, TimeMachine only adds the starting state to the state corpus.

TimeMachine is an automatic time-travelling-enabled test generator for Android apps that implements several heuristics to choose the most progressive state from the state corpus to explore next. Intuitively, a state reaching which covered new code and that has been difficult to reach has more potential to trigger new program behavior. TimeMachine dynamically collects such feedback to identify the most progressive state. TimeMachine identifies a progressive state as one which itself was infrequently visited *and* the k nearest neighbors⁴ were visited relatively infrequently.

Our experiments demonstrate a substantial performance increase over our baseline test generation tool—Monkey extended with system-level event generator of Stoa [40]. Given the 68 apps in the AndroTest benchmark [23], our time-travel strategy enables the baseline tool to achieve 1.15 times more statement coverage and to discover 1.73 times more unique crashes. Given 37 apps

in the benchmark of industrial apps, around 900 more methods are covered on average and 1.5 times more unique crashes are discovered. Our time-travel strategy makes TimeMachine so efficient that it outperforms the state-of-the-art test generators Sapienz [36] and Stoa [40] both in terms of coverage as well as errors found, detecting around 1.5 times more unique crashes than the next best test generator. TimeMachine tested the Top-100 most popular apps from Google Play and found 137 unique crashes.

In summary, our work makes the following contributions:

- We propose time-travel testing for Android which resumes the most progressive states observed in the past so as to maximize efficiency during the exploration of an app’s state space. The approach identifies and captures interesting states as save points, detects when there is a lack of progress, and resumes the most progressive states for further testing. For instance, it can quickly deprioritize the main screen state which is visited by most sequences, and resume/test difficult-to-reach states. We propose several heuristics that guide execution to a progressive state when progress is slow.
- We implement the time-travel testing framework and an automated, feedback-guided, time-travel-enabled state space exploration technique for Android apps. The framework and testing technique are evaluated on both open-source and closed-source Android app benchmarks, as well as top-100 popular apps from Google Play. We have made our time-travel Android app testing tool TimeMachine publicly available on Github: <https://github.com/DroidTest/TimeMachine>

2 TIME-TRAVEL FRAMEWORK

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 2 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.

³Named after the celebrated fictional work by H.G. Wells more than a century ago.

⁴The k nearest neighbors are states reachable along at most k edges.

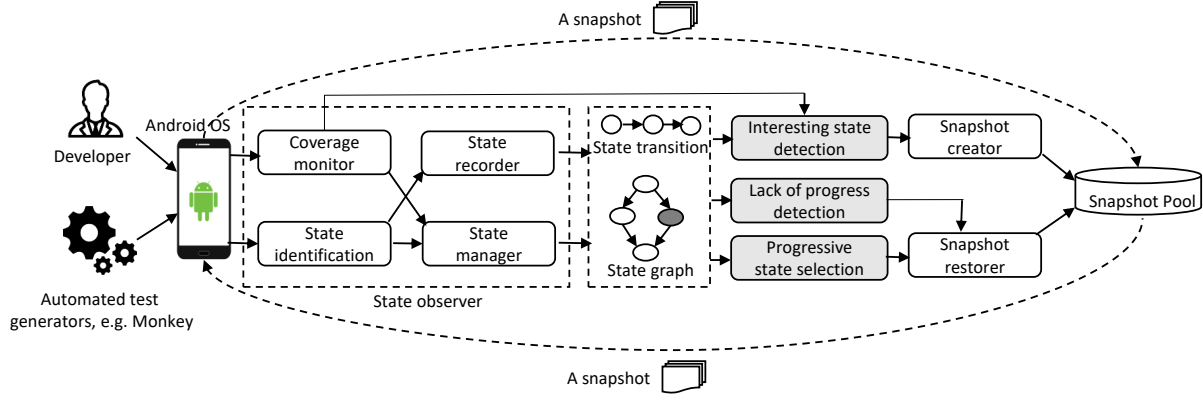


Figure 2: Time travel framework. Modules in grey are configurable, allowing users to adjust strategy according to scenarios.

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 [21, 22].

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 [9]. For closed-source, industrial apps, we collect method coverage using the Ella coverage tool [8]. 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.

3 METHODOLOGY

We develop the first time-travel-enabled test generator TimeMachine for Android apps by enhancing Android Monkey [3] with our framework. TimeMachine’s procedure is presented in Algorithm 1. TimeMachine’s objective is to maximize state and code coverage. TimeMachine starts with a snapshot of the initial state (lines 1-4). For each event that Monkey generates, the new state is computed and the state transition graph updated (lines 5-9). If the state is INTERESTING (Sec. 3.1), a snapshot of the VM is taken and associated with that state (lines 10-13). If Monkey is STUCK and no more progress is made (Sec. 3.2), TimeMachine finds the most progressive state (SELECTFITTESTSTATE; Sec. 3.3) and restores the associated VM snapshot (lines 14-17). Otherwise, a new event is generated and loop begins anew (lines 5-18).

3.1 Identifying Interesting States

TimeMachine identifies an interesting state based on changes in GUI or code coverage (Line 10 in Algorithm 1). The function `ISINTERESTING(state)` returns true if (1) *state* is visited for the first time, and (2) when *state* was first reached new code was executed.

Algorithm 1: Time-travel testing (TimeMachine).

Input: Android App, Sequence generator Monkey

- 1: State $curState \leftarrow \text{LAUNCH}(App)$
- 2: Save VM snapshot of $curState$
- 3: Interesting states $states \leftarrow \{curState\}$
- 4: State Transition Graph $stateGraph \leftarrow \text{INITGRAPH}(curState)$
- 5: **for each** Event e in $Monkey.GENERATEEVENT()$ **do**
- 6: **if** timeout reached **then break; end if**
- 7: $prevState \leftarrow curState$
- 8: $curState \leftarrow \text{EXECUTEEVENT}(App, e)$
- 9: $stateGraph \leftarrow \text{UPDATEGRAPH}(prevState, curState)$
- 10: **if** $\text{ISINTERESTING}(curState, stateGraph)$ **then**
- 11: Save VM snapshot of $curState$
- 12: $states \leftarrow states \cup \{curState\}$
- 13: **end if**
- 14: **if** $\text{ISSTUCK}(curState, stateGraph)$ **then**
- 15: $curState \leftarrow \text{SELECTFITTESTSTATE}(states, stateGraph)$
- 16: Restore VM snapshot of $curState$
- 17: **end if**
- 18: **end for**

Output: State Transition Graph $stateGraph$

The intuition behind our definition of “interesting” states is that the execution of new code provides the evidence that a functionality that has not been tested before is enabled in the discovered state. More new code related to the functionality might be executed by exploring this state. For instance, suppose clicking a button on screen S_1 leads to a new screen S_2 , from where a new widget is displayed (increasing code coverage). The new widget comes with its own event handlers that have not been executed. These event handlers can be covered by further exploring screen S_2 . This heuristic not only accurately identifies an interesting state (S_2 in this case) but also significantly reduces the total number of saved states (since only interesting states are saved during testing).

3.2 Identifying Lack of Progress

The testing process can stay unprogressive without discovering any new program behavior for quite some time. As *reasons* for Monkey getting stuck, we identified loops and dead ends.

Loops. A loop is observed when the same few (high-frequency) states are visited again and again. To easily perform routine activities, app pages are typically organized under common patterns, e.g., from the main page one can reach most other pages. This design leads to a phenomenon where random events tend to trigger transitions to app pages which are easy to trigger. Moreover, apps often browse nested data structures, it is difficult to jump out from them without human knowledge. For example, let us consider the AnyMemo [7] app, a flashcard learning app we tested. Monkey clicks a button to load a CSV file and arrives at an app page that browses system directories. It keeps on exploring directories and cannot leave this app page until it finds a CSV file to load (or by pressing the “Back” button many times in a row). In our experiments, Monkey could not jump out of the loop within 5000 events.

Algorithm 2: Detecting loops and dead-ends (ISSTUCK).

Input: Queue length l

Input: Lack-of-progress threshold $maxNoProgress$

Input: Max. top $(\alpha \cdot 100)\%$ most frequently visited states

Input: Max. proportion β of repeated plus frequent states

- 1: FIFO Queue \leftarrow empty queue of length l
- 2: $noProgress = 0$ // #events since last state transition
- 3:
- 4: **procedure** ISSTUCK(State $curState$, Graph $stateGraph$) {
- 5: $prevStateID = \text{Queue.TOP}()$
- 6: **if** $prevStateID == curState.ID$ **then**
- 7: $noProgress \leftarrow noProgress + 1$
- 8: **else**
- 9: $\text{Queue.PUSH}(curState.ID)$
- 10: $noProgress = 0$
- 11: **end if**
- 12: **if** $noProgress > maxNoProgress$ **then**
- 13: **return true** // detect dead ends
- 14: **end if**
- 15: **if** $\text{Queue.LENGTH} == l$ **then**
- 16: $nRepeated \leftarrow \text{COUNTMAXREPEATEDSTATES}(\text{Queue})$
- 17: $nFrequent \leftarrow \text{COUNTFREQSTATES}(\text{Queue}, stateGraph, \alpha)$
- 18: **if** $(nRepeated + nFrequent)/l > \beta$ **then**
- 19: **return true** // detect loops
- 20: **end if**
- 21: **end if**
- 22: **return false**
- 23: }

Dead ends. A dead end is a state which is difficult to exit. Some pages require specific inputs which are very unlikely to be randomly generated. Monkey can be trapped by them and can keep on generating events without making any “progress”. For instance, consider an app page in AnyMemo [7] where a form needs to be filled and submitted. Yet, the “Submit” button is located at the bottom of the page, and does not even appear on screen. Monkey would need to correctly fill in certain parameters, scroll all the way to the bottom, and then generate a “Click” event on the button to transition to exit the page. This is quite unlikely. Monkey gets stuck in a dead end.

When TimeMachine gets stuck, the most progressive state is traveled back to (lines 14-17 in Algorithm 1). The function ISSTUCK is sketched in Algorithm 2 and realizes a sliding window algorithm. Firstly, four parameters must be specified, which are explained later. There are two global variables, a queue of *specified* length l and a counter which keeps track how often the same state has been observed (lines 1-3). Given the current app state and the state transition graph, if the current state is the same as the previous state the no-progress counter is incremented (lines 4-7). Otherwise, the counter is reset (lines 8-11). If the counter exceeds the *specified* maximum ($maxNoProgress$), then a dead end is detected (lines 12-14). If the fixed-length queue is filled and the proportion of “easy” states in the queue surpasses the *specified* threshold β , then a loop is detected. Two kinds of states in the queue are considered *easy*: states occurring multiple times in the queue, and states among the top α percentage of the most frequently visited states.

Algorithm 3: Selecting the next state (SELECTFITTESTSTATE)**Input:** Path length k

```

1: procedure SELECTFITTESTSTATE( $states, stateGraph$ ) {
2:    $bestFitness \leftarrow 0$ 
3:   for each  $state$  in  $states$  do
4:      $stateFitness \leftarrow 0$ 
5:      $paths \leftarrow$  all paths in  $stateGraph$  of length  $k$  from  $state$ 
6:     for each  $path$  in  $paths$  do
7:       for each Node  $s$  in  $path$  do
8:          $stateFitness \leftarrow stateFitness + f(s)$  // see Eq. (1)
9:       end for
10:    end for
11:     $stateFitness \leftarrow \frac{stateFitness}{|paths|}$ 
12:    if  $stateFitness > bestFitness$  then
13:       $bestState = state$ 
14:       $bestFitness = stateFitness$ 
15:    end if
16:  end for
17:  return  $bestState$ 
18: }
```

3.3 Progressive State Selection

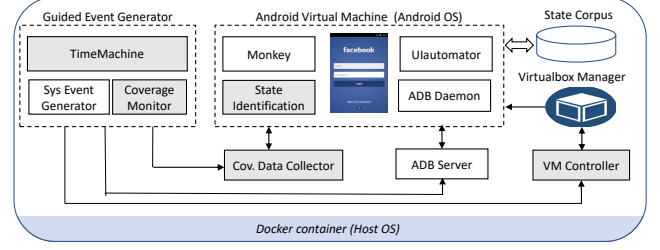
In order to select a state to travel back to once Monkey is **STUCK**, we assign a fitness to each state which evaluates its potential to trigger new program behavior (lines 14–17 in Alg. 1). The fitness $f(s)$ of a state s is determined by the number of times the state has been visited and the number of “interesting” states generated from it. Concretely, the fitness function is defined as:

$$f(s) = f_0 * (1 + r)^{w(s)} * (1 - p)^{v(s) - w(s)} \quad (1)$$

where $v(s)$ is the number of times state s is visited and $w(s)$ is the number of “interesting states” generated from state s ; r is a reward of finding an interesting state and p is a penalty of transiting to a state that has already been discovered; f_0 is the initial value. In TimeMachine, the initial value of an interesting state is set as 6 times of that of an uninteresting state, and r as well as p are set as 0.1. When a state is repeatedly being visited and no interesting states are discovered, its fitness keeps on being reduced due to penalty p so that other state will be selected and restored eventually.

Maximizing benefit of time travel. The definition of state fitness in Equation (1) does not account for the fact that events executed on that state may quickly trigger a departure from that state, again advancing through unprogressive states. To maximize benefit of time-travel, we develop an algorithm that selects the state with a high-fitness “neighborhood”, i.e., the state which has neighboring states which also have a high fitness.

Algorithm 3 outlines the process of selecting the most progressive state for time travel. It takes as input the interesting states that have an associated VM snapshot and the state transition graph that is maintained by our time-travel framework. The number of transitions k which determines a state’s “neighborhood” must be specified by the user. In our experiments, we let $k = 3$. For each interesting $state$, TimeMachine computes the average fitness of a state in the k -neighborhood of the state. The state with the maximum average state fitness in its k -neighborhood is returned. The

**Figure 3:** Architecture of TimeMachine implementation.

k -neighborhood of $state$ are all states s in $stateGraph$ that are reachable from $state$ along at most k transitions. The fitness $f(s)$ of a state s is computed according to Equation (1). With this algorithm, Monkey not only travels in time to the state with the highest fitness value but also continues to explore states with high fitness values within k transitions, which maximizes the benefit of time travel.

4 IMPLEMENTATION

Our time travel framework is implemented as a fully automated app testing platform, which uses or extends the following tools: VirtualBox [4], the Python library pyvbox [11] for running and controlling the Android-x86 OS [6], Android UI Automator [10] for observing state transitions, and Android Debug Bridge (ADB) [5] for interacting with the app under test. Figure 3 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 [9] (statement coverage) and closed-source apps using Ella [8] (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. To facilitate the analysis of all benchmarks, we integrated TimeMachine with two Android versions. TimeMachine works with the most widely-used version, Android Nougat with API 25 (Android 7.1). However, to perform end-to-end comparison on AndroTest benchmark [23], we also implement TimeMachine on Android KitKat version with API 19 (Android 4.4). The publicly available version of Sapienz [36] (a state-of-the-art/practice baseline for our experiments) is limited to Android API 19 and cannot run on 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 Stoa [40] to support system events such as phone calls and SMSs.

5 EMPIRICAL EVALUATION

In our experimental evaluation, we seek to answer the following research questions.

RQ1 How effective is our time-travel strategy in terms of achieving more code coverage and finding more crashes? We compare TimeMachine to the baseline into which it was implemented.

- RQ2** How does time-travel testing (i.e., TimeMachine) compare to state-of-the-art techniques in terms of achieved code coverage and found crashes?
- RQ3** How does time-travel testing (i.e., TimeMachine) perform on larger, real-world apps, such as industrial apps and Top-100 apps from Google Play?

5.1 Experimental Setup

To answer these research questions, we conducted three empirical studies on both open-source and closed-source Android apps.

Study 1. To answer RQ1, we evaluate TimeMachine and baseline tools on AndroTest [23] and investigate how achieved code coverage and found faults are improved by using the time-travel strategy. We chose AndroTest apps as subjects because AndroTest has become a standard testing benchmark for Android and has been used to evaluate a large number of Android testing tools [16, 20, 23, 34–37, 40, 43]. It was created in 2015 by collecting Android apps that have been used in evaluations of 14 Android testing tools.

TimeMachine applies time-travel strategy to a baseline tool; *the baseline tool is Monkey extended with Stoa’s system-level event generator*. To accurately evaluate effectiveness of time-travel strategy, we set Monkey extended with the system-level event generator from Stoa as baseline (called *MS*). We chose *MS* instead of Monkey as a baseline tool to make sure that the improvement achieved by TimeMachine completely comes from time-travel strategy, not from system event generation.

We also implement another variant of Monkey as baseline to evaluate effectiveness of “heavy components” such as state saving and restoring on enhancing a test technique. This variant applies only the lack of progress detection component of our time-travel strategy without state saving and restoring components. When lack of progress is detected, it simply restarts testing from scratch, i.e., re-launching app under test without resuming states (called *MR*).

In TimeMachine, parameters l , $maxNoProgress$, α , β for *ISSTUCK* in Alg. 2 are set to 10, 200, 0.2, and 0.8, respectively. These values were fixed during initial experiments of two authors with three apps from AndroTest (Anymemo, Bites, aCal). We executed these apps with Monkey for many rounds and recorded relevant data such as the number of state transitions when a loop was observed and the number of executed events when Monkey jumped out from a dead end. Based on observed data and authors’ heuristics, we came up with several groups of values and evaluated them on these three apps, and eventually chose above data as default parameter values. In the evaluation, TimeMachine used the default values for all the three studies. Baseline tool *MS* and *MR* use the same parameter values as in TimeMachine.

Study 2. To answer RQ2, we evaluate TimeMachine and state-of-the-art app testing tools on AndroTest and compare them in terms of achieved code coverage and found crashes. For state-of-the-art tools, we chose Monkey [3], *Sapienz* [36], and *Stoa* [40]. Monkey is an automatic random event sequence generator for testing Android apps and has been reported to achieve the best performance in two works [23, 42]. *Sapienz* and *Stoa* are the most recent techniques for Android testing. These testing tools have also been adequately tested and are standard baselines in the Android

testing literature. To have a fair comparison, all techniques use their default configuration.

Study 3. To answer RQ3, we evaluate TimeMachine, baseline tools and all state-of-the-art techniques on large real-world Android apps, and investigate whether they have a consistent performance on both closed-source and open-source Android apps. In this evaluation, we use *IndustrialApps* [42] as subject apps. *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 [8]. 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).

To further investigate the usability of TimeMachine, we evaluate TimeMachine on Top-100 popular Android apps from Google Play and investigate whether TimeMachine can effectively detect crashes in online apps, i.e., those available for download from Google Play at the time of writing. Following the practice adopted by some previous authors [36, 40] of applying the technique to top popular apps on Google Play, we focus on analyzing detected crashes by TimeMachine and do not compare TimeMachine with state-of-the-art techniques on this data set. Top-100 popular apps were collected by downloading the most highly ranked apps on Google Play and instrumenting them with our coverage tool Ella until we obtained 100 apps that could be successfully instrumented by Ella.

Procedure. To mitigate experimenter bias and to scale our experiments, we chose to provide no manual assistance during testing in all studies. For all test generators, the Android testing is fully automatic. None of the test generators is seeded with an initial set of event sequences. The testing process is automatically started after installation. All data are generated and processed automatically. We neither provide any input files, nor create any fake accounts. Each experiment is conducted for six (6) hours and repeated five (5) times totalling 35580 CPU hours (≈ 4.1 year). To mitigate the impact of random variations during the experiments, we repeated each experiment five times and report the average. In comparison, the authors of *Sapienz* report one repetition of one hour while the authors of *Stoa* report on five repetitions of three hours. We chose a time budget of six hours because we found that the asymptotic coverage was far from reached after three hours in many apps (i.e., no saturation had occurred).

Coverage & Crashes. We measure code coverage achieved and errors discovered within six hours. To *measure statement or method coverage*, we use Emma and Ella, the same coverage tools that are used in *Sapienz* and *Stoa*. To *measure the number of unique crashes detected*, we parse the output of Logcat,⁵ an ADB tool that dumps a log of system messages. We use the following protocol to identify a unique crash from the error stack (taken from Su et al. [40]):

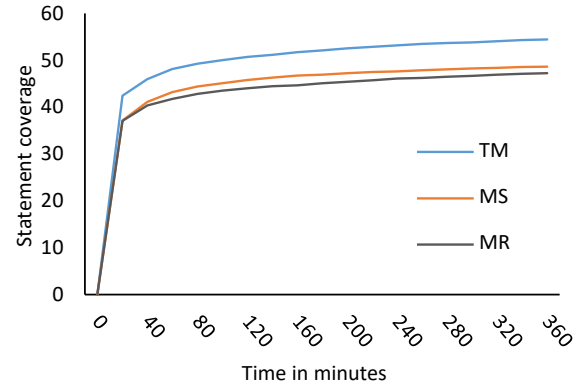
- Remove all unrelated crashes by retaining only exceptions containing the app’s package name (and filtering others).

⁵<https://developer.android.com/studio/command-line/logcat>

Table 1: Results from AndroTest (68 open-source apps).

Subjects	TimeMachine			Baselines				State-of-the-art					
	%Cov	#Crash	#State	%Coverage		#Crashes		%Coverage		#Crashes			
	-	-	-	MS	MR	MS	MR	ST	SA	MO	ST	SA	MO
A2DP	46	1	222	42	35	0	0	47	44	39	1	4	0
aagtl	19	5	34	16	16	3	3	17	19	17	3	6	3
Aarddict	19	2	31	18	14	1	0	37	15	13	5	0	0
aCal	29	9	178	28	26	7	1	22	27	18	7	5	3
addi	19	2	63	19	19	3	4	14	20	19	3	1	4
adsdroid	39	2	23	32	36	4	2	31	36	30	2	1	1
aGrep	64	3	77	55	57	2	3	37	59	46	1	2	1
aka	83	1	166	62	77	2	1	81	83	65	1	8	1
alarmclock	65	4	41	69	65	5	0	65	75	70	3	5	5
aLogCat	81	0	114	74	65	0	0	-	-	63	-	-	0
Amazed	67	1	25	62	40	0	1	57	66	36	0	2	1
anycut	68	0	37	63	67	0	0	59	65	63	0	0	0
anymemo	47	12	311	40	36	5	2	36	53	32	7	7	2
autoanswer	23	3	45	19	17	1	0	20	16	13	2	0	0
batterydog	67	2	32	63	70	1	1	54	67	64	1	1	0
battery	83	13	58	76	80	16	5	75	78	75	1	18	0
bites	49	8	68	37	40	5	0	38	41	37	2	1	1
blockish	73	0	71	50	49	0	0	36	52	59	0	2	0
bomber	83	0	32	80	80	1	0	57	75	76	0	0	0
Book-Cat	30	7	109	28	29	0	1	14	32	29	3	2	1
CDT	81	0	49	79	66	0	0	79	62	77	0	0	0
dalvik-exp	73	7	65	69	72	7	3	70	72	68	6	2	2
dialer2	47	3	47	38	51	0	0	33	47	38	3	0	0
DAC	88	2	39	85	88	0	0	53	83	86	0	5	0
fileexplorer	59	0	32	41	55	0	0	41	49	41	0	0	0
fbubble	81	0	15	81	81	0	0	50	76	81	0	0	0
gestures	55	0	29	36	55	0	0	32	52	36	0	0	0
hndroid	20	6	39	10	8	2	1	9	15	8	1	2	1
hotdeath	76	2	61	81	69	1	0	60	75	76	1	2	1
importcont	43	1	51	41	40	0	0	62	39	40	0	0	1
Jamendo	58	8	107	57	57	6	1	44	63	55	7	3	0
k9mail	9	15	50	8	8	16	1	8	7	7	16	2	1
LNM	-	-	-	-	-	-	-	-	-	-	-	-	-
LPG	80	0	101	76	77	0	0	68	79	76	0	0	0
LBBuilder	31	1	102	28	28	0	0	25	27	27	0	0	0
manpages	75	1	189	69	72	0	0	63	73	39	1	0	0
mileage	52	21	136	46	43	14	2	39	49	39	13	9	3
MNV	46	6	292	39	42	0	2	45	63	40	3	0	3
Mirrored	66	9	88	45	46	1	0	51	59	59	6	8	5
multisms	66	1	68	58	59	0	0	45	61	33	1	0	0
MunchLife	77	0	47	67	75	0	0	65	72	69	0	0	0
MyExp	54	1	109	51	49	0	0	48	60	42	1	1	0
myLock	47	3	118	45	29	2	0	44	31	27	2	0	0
nectroid	62	3	53	36	34	0	0	64	66	33	3	1	0
netcounter	63	3	60	57	58	1	0	70	70	42	2	1	0
PWMG	58	4	56	50	43	1	3	62	58	53	6	4	3
PWM	18	0	80	12	7	0	0	6	8	7	0	0	0
Photos	38	2	29	29	24	1	2	30	34	30	2	1	1
QSettings	51	0	256	48	45	0	0	42	52	51	0	1	0
RMP	62	1	34	52	58	0	0	70	58	53	1	0	0
ringdroid	52	3	63	23	50	0	1	-	38	22	-	1	0
sanity	33	3	407	35	28	2	3	27	21	27	2	3	2
soundboard	63	0	27	42	61	0	0	42	51	42	0	0	0
SMT	79	0	34	80	38	0	0	80	80	18	0	0	0
SpriteText	61	0	23	59	60	0	0	59	60	59	0	0	0
swiftp	14	0	42	13	14	0	0	13	14	13	0	0	0
SyncMyPix	25	1	66	25	20	0	0	25	21	20	1	1	0
tippytipper	81	0	112	77	80	0	0	77	83	79	0	0	0
tomdroid	56	0	68	55	50	0	0	54	56	48	1	1	0
Translate	51	0	41	37	49	0	0	44	49	48	0	0	0
Triangle	-	-	-	-	-	-	-	-	-	-	-	-	-
wchart	71	0	92	69	63	0	0	47	73	64	2	6	0
WHAMS	71	1	68	61	69	0	0	72	77	64	1	0	0
wikipedia	36	0	204	33	35	0	0	30	32	34	0	0	0
Wordpress	8	12	52	8	4	4	2	8	6	4	12	2	0
worldclock	92	1	99	90	91	0	0	92	91	90	1	0	0
yahtzee	60	2	46	48	33	0	0	60	58	52	1	0	2
zooborns	38	1	30	35	37	1	0	33	36	35	2	0	0
Ave/Sum	54	199	85	47	47	115	45	45	51	44	140	121	48

- Given the related crash information, extract only the crash stack and filter out all information that is not directly relevant (e.g., the message “invalid text...”).
- Compute a hash over the sanitized stack trace of the crash to identify unique crashes. Different crashes should have a different stack trace and thus a different hash.

**Figure 4: Progressive statement coverage for TimeMachine (TM) and baseline tools on 68 benchmark apps.**

Execution environment. The experiments were conducted on two physical machines with 64 GB of main memory, running a 64-bit Ubuntu 16.04 operating system. One machine is powered by an Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz with 56 cores while the other features an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz with 40 cores. To allow for parallel executions, we run our system in Docker (v1.13) containers. Each Docker container runs a VirtualBox (v5.0.18) VM configured with 2GB RAM and 2 cores for the Android 4.4 and 2 cores and 4GB RAM for Android 7.1. We made sure that each evaluated technique is tested under the same workload by running all evaluated techniques for the same app on the same machine.

5.2 Experimental Results

5.2.1 Study 1: Effectiveness of Time-travel Strategy.

Table 1 shows achieved coverages and found faults by each technique on 68 Android apps. The highest coverage and most found crashes are highlighted with the grey color for each app. The results of TimeMachine and baseline techniques are shown in column “TimeMachine ” and “Baselines”. Recall that MS indicates Monkey extended with Stoa’s system-level event generator, and MR indicates Monkey with the ability to restart testing from scratch when lack of progress is detected.

Comparison between TimeMachine and MS. TimeMachine achieves 54% statement coverage on average and detects 199 unique crashes for 68 benchmark apps. MS 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 MS. To further investigate these results, Figure 4 presents achieved code coverage over execution time for all 68 apps. As we can see, TimeMachine has achieved higher coverage from around the 20th minute onwards, finally achieving 7% more statement coverage at the end of execution time. Figure 5 presents the box-plots of the final coverage results for apps grouped by size-of-app, where “x” indicates the mean for each box-plot. We see that coverage improvement is substantial for all four app size groups.

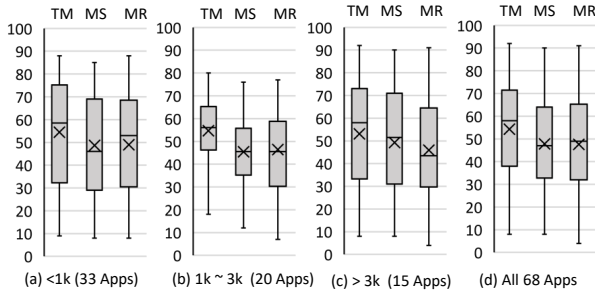


Figure 5: Statement coverage achieved by TimeMachine (TM), MS and MR.

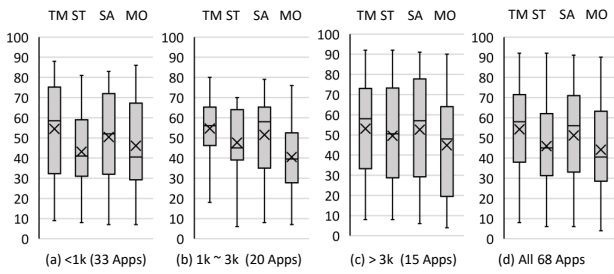


Figure 6: Statement coverage achieved by TimeMachine (TM), Stoa (ST), Sapienz (SA) and Monkey (MO).

Our time-travel strategy effectively enhances the existing testing technique (MS) by achieving 1.15 times statement coverage and detecting 1.73 times crashes on 68 benchmark apps.

Comparison between TimeMachine and MR. MR achieves 47% statement coverage on average and detects 45 unique crashes for 68 benchmark apps. TimeMachine achieves 1.15 times statement coverage and 4.4 times unique crashes more than MR. Similarly, Figure 4 and Figure 5 shows TimeMachine covers more code in a short time and substantially improves statement coverage for all four app size groups compared to MR. This shows that it is not sufficient to simply restart an app from scratch when lack of progress is detected, though MR improves Monkey by 3% statement coverage (Monkey’s statement coverage is shown in the third subcolumn of column “State-of-the-art” of Table 1).

State saving and restoring as well as other components substantially contribute to enhancing testing techniques, it is not sufficient to simply restart app from scratch when lack of progress is detected.

5.2.2 Study 2: Testing Effectiveness.

The results of state-of-the-art techniques are shown in column “State-of-the-art” of Table 1 (ST, SA, and MO indicate Stoa, Sapienz and Monkey, respectively). As can be seen, TimeMachine achieves the highest statement coverage on average (54%) and is followed by Sapienz (51%), Stoa (45%) and Monkey (44%). Figure 6 also shows that TimeMachine achieves the highest statement coverage for all

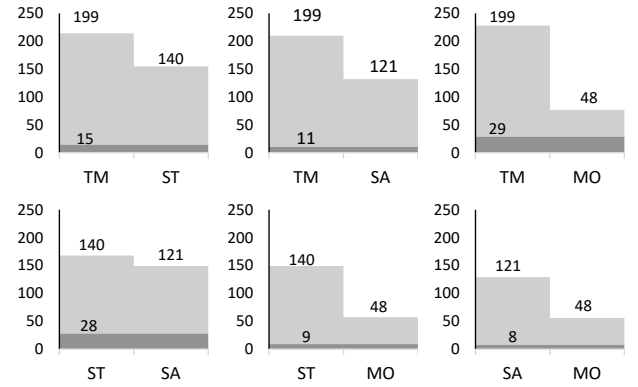


Figure 7: Comparison of total number of unique crashes for AndroTest apps. The dark grey areas indicate the proportion of crashes found by both techniques.

four app size groups. TimeMachine detects the most crashes (199) as well, followed by Stoa (140), Sapienz (121) and Monkey (48).

The better results from TimeMachine can be explained as follows: state-level feedback accurately identifies which parts in app are inadequately explored. Moreover an inadequately explored state can be arbitrarily and deterministically launched for further exploration via restoring a snapshot. Existing techniques typically observe program behavior over an event sequence that often is very long and goes through many states. Coverage feedback of an individual state is unavailable. So our time travel framework enhances app testing by providing fine-grained state-level coverage feedback.

TimeMachine achieves the highest statement coverage and detects the most crashes on 68 benchmark apps compared to state-of-the-art techniques. Promisingly, our time-travel framework has a potential to enhance state-of-the-art app testing tools to achieve better results.

To study performance across apps, for each technique under evaluation, we compute the number of apps on which the technique achieves the best performance. In terms of statement coverage, TimeMachine achieves the best performance on 45 apps, followed by Sapienz (19 apps), Stoa (11 apps) and Monkey (1 app). For detected crashes, TimeMachine achieves the best performance on 32 apps. For Stoa, Sapienz, and Monkey, there are 16, 15, and 4 apps, respectively. We further perform a pairwise comparison of detected crashes among evaluated techniques. As shown in Figure 7, there is less than ten percent overlap between the crashes found by TimeMachine and Stoa, or TimeMachine and Sapienz, respectively. The overlap with Monkey is reasonably high. About 60% of unique crashes found by Monkey are also found by TimeMachine; however TimeMachine found many new crashes which are not found by Monkey. This analysis shows that TimeMachine can be used together with other state-of-the-art Android testing techniques to jointly cover more code and discover more crashes.

TimeMachine complements state-of-the-art Android testing techniques in terms of the ability to discover more crashes and cover more code.

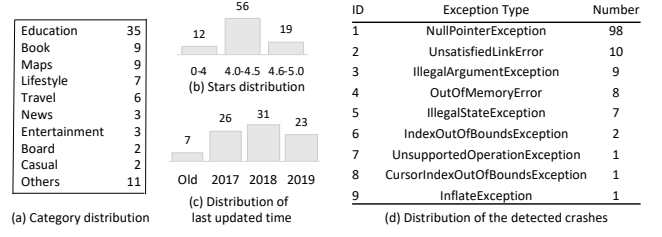
Table 2: Results from 37 closed-source industrial apps.

Subject		%Coverage					#Crashes					#State
Name	#Method	TM	ST	MO	MS	MR	TM	ST	MO	MS	MR	TM
AutoScout24	49772	34	25	29	31	29	18	0	1	12	0	915
Best Hairstyles	28227	14	20	13	15	14	1	0	0	1	0	34
Crackle	48702	22	9	19	22	22	21	0	8	10	8	905
Duolingo	46382	26	13	22	23	22	12	0	0	8	0	384
ES File Explorer	47273	28	15	18	24	21	9	0	0	8	0	594
Evernote	45880	11	10	7	11	8	0	0	0	0	0	45
Excel	48849	19	9	14	14	14	2	0	0	0	0	201
Filters For Selfie	17145	9	13	8	9	8	0	0	0	0	0	43
Flipboard	41563	30	17	24	28	25	0	0	0	0	0	308
Floor Plan Creator	23303	29	30	23	26	26	0	0	0	0	0	394
Fox News	42569	28	13	21	20	17	5	0	1	4	0	635
G.P. Newsstand	50225	7	6	5	6	5	0	0	0	0	0	14
GO Launcher Z	45751	13	9	11	11	12	0	0	0	0	0	81
GoodRx	48222	24	19	21	22	21	17	0	0	11	0	468
ibisPaint X	47721	16	16	10	12	12	0	1	0	0	1	655
LINE Camera	47295	17	15	14	15	15	22	1	1	19	1	413
Marvel Comics	43578	18	18	15	17	15	0	0	0	0	0	133
Match	50436	15	11	11	15	11	0	0	0	0	0	106
Merriam-Webster	50436	25	21	18	24	23	6	0	2	3	2	1018
Mirror	36662	8	8	8	8	8	0	0	0	0	0	23
My baby Piano	20975	7	7	7	7	7	0	0	0	0	0	4
OfficeSuite	45876	17	11	16	16	16	3	0	0	1	0	479
OneNote	50100	11	11	11	10	11	23	0	4	9	2	181
Pinterest	46071	21	10	16	15	8	0	0	0	0	0	382
Quizlet	48369	33	22	30	33	31	23	0	0	17	0	548
Singing	46521	10	6	5	8	6	14	0	0	12	0	77
Speedometer	47773	16	11	13	13	15	0	0	0	0	0	51
Spotify	44556	13	14	10	11	10	0	0	0	0	0	36
TripAdvisor	46617	26	22	21	23	22	1	1	0	0	0	1279
trivago	50879	16	9	8	14	10	7	0	0	6	0	139
WatchESPN	43639	26	22	23	25	24	0	0	0	0	0	395
Wattpad	44069	25	14	13	22	13	0	0	0	0	0	327
WEBTOON	47465	21	17	13	19	16	12	0	0	8	1	487
Wish	48207	16	15	17	15	14	4	0	0	4	0	55
Word	49959	16	9	14	14	14	0	0	0	0	0	146
Yelp	46903	24	16	17	20	17	69	0	0	37	0	395
Zedge	46799	24	23	22	23	21	12	0	3	13	0	911
Ave/Sum	44182	19	14	15	17	15	281	3	20	183	15	358

5.2.3 Study 3: Closed-source Apps.

Table 2 shows results of 37 closed-source benchmark apps. It is clear that 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. Note that the improvement of 2% to 4% is considerable since each app has 44182 methods on average and around 900 to 1800 more methods are covered for each app. In terms of number of crashes found, TimeMachine detects 1.5 times and 18.7 times crashes more than MS and MR, respectively. MS detects 183 crashes and MR detects 15 crashes.

Compared to state-of-the-art techniques, TimeMachine substantially outperforms them on both method coverage and the number of found crashes. Stoa achieves 14% method coverage and detects 3 crashes. Monkey achieves 15% method coverage and 20 crashes. Unexpectedly, Stoa demonstrated the worst results, worse than Monkey. A closer inspection revealed that these real-world apps use complex UI containers (e.g., animations), which pose difficulties for Stoa to build a model. Certain app pages might be missed entirely because the event handlers associated with those widgets cannot be triggered. However, both TimeMachine and Monkey overcome this issue due to their independence from any UI models. We reported this matter to the authors of Stoa who confirmed our insight.

**Figure 8: Statistics of tested 87 apps from Google Play.**

Our time-travel strategy substantially improve the existing technique (i.e., MS) by covering around 900 more methods and discovering 1.5 times more crashes. TimeMachine also outperforms state-of-the-art techniques (Stoa and Monkey) in terms of both method coverage and the number of found crashes.

Out of Top-100 instrumented apps from Google Play, we successfully tested 87 apps. The remaining 13 apps kept crashing due to a self-protection-mechanism (though they were successfully instrumented). As shown in Figure 8, the tested apps are quite diverse being selected from more than 10 categories. It comes as no surprise that the majority of them are ranked with over 4 stars, and are being actively maintained.

In the 87 apps, we found 137 unique crashes. These are all non-native crashes, i.e., their stack traces explicitly point to the source line of the potential faults in the tested app. The detected 137 crashes were caused by 9 kinds of exceptions shown in Figure 8. The most common type is NullPointerException.

In total, TimeMachine detects 137 unique crashes in 25 of Top-100 Google Play apps.

5.2.4 Analysis on State Identification and Time-travel.

State identification. The evaluation shows GUI layout is appropriate to identify an app state. This state abstraction generates acceptable number of states for an app, at the same time sufficiently captures features of a state. As we see column "#State" in Table 1 and Table 2, more states are discovered in apps with rich functionality and less states are discovered in simple apps. For instance, app AnyMemo with plentiful activities has 311 found states and app Frozenbubble with few activities has 15 found states. Similar results can be found for industrial apps. Note that no login is provided in the evaluation, such that TimeMachine might identify a small number of states for some large scale apps, like K9Mail.

GUI layout sufficiently captures features of an app state. On average, 85 states are found in an open source benchmark app and 358 states are found in an industrial benchmark app.

Frequency. An automatic Android testing tool that generates a very long event sequence, like Monkey, may get stuck in loops or dead ends. We measure the number of times, our time-travelling infrastructure is employed to understand how often Monkey gets stuck. In six hours over all runs and apps, Monkey gets stuck with a mean of 305.6 and a median of 308.5 times. As we can see in the box plots of Figure 9, there are generally less restores as the

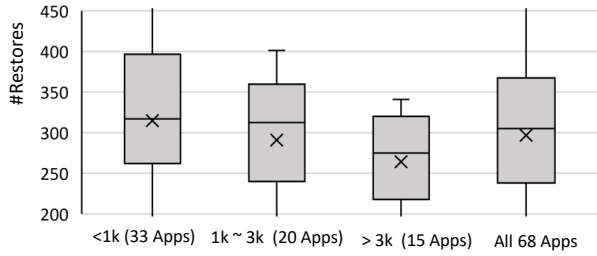


Figure 9: Boxplots. Depending on app size how often does TimeMachine travel back in time?

apps get bigger. This is due to an increasing state space. A greater proportion of random actions lead to yet undiscovered states such that progress is maintained. Smaller apps have a small state space and Monkey may run more quickly into loops of states that have already been observed.

On average, TimeMachine travels about 51 times per hour back to more progressive states that were observed in the past—because Monkey gets stuck in a loop or dead end.

Cost. The cost of time-travel is acceptable. TimeMachine spends around 10 seconds taking a snapshot and 9 seconds restoring a snapshot on an Android7.1 virtual machine with 4 GB memory. A snapshot takes around 1 GB disk space. For large scale industrial apps in the evaluation, a session typically generates less than 100 snapshots. So TimeMachine is able to run on a desktop in term of consumed storage space. Besides, since one snapshot is stored for each "interesting" state, storage can be potentially reduced by re-configuring the definition of "interesting".

This is a reasonable cost for reaching a particular state in a deterministic way for Android testing, especially for large scale apps. To reach a deep state, a human tester may need to perform dozens of events and repeat them many times due to non-determinism of Android apps. This is even more difficult for an automated test generators because it typically requires generating a very long event sequence automatically. These tools thus spend more time reaching hard-to-reach states than TimeMachine, which makes reachability easier by recording and restoring snapshots of "interesting" states.

The cost of time-travelling is acceptable, and also reasonable for testing stateful programs.

6 THREATS TO VALIDITY

We adopted several strategies to enhance *internal validity* of our results. To mitigate risks of selection bias, we chose apps in a standard testing benchmark which has been used in previous studies [16, 20, 23, 34–37, 40, 43]. In order to put no testing tool at a disadvantage, we used default configurations, provided the exact same starting condition, and executed each tool several times and under the same workload. To identify unique crashes, we followed the Stoa protocol [40] and also manually checked the crashes found. To measure coverage, we used a standard coverage tool.

We realise that our results on Stoa versus Sapienz and those reported in the Stoa paper [40] are vastly different. We checked with the authors of Stoa [40] on this matter. The authors of Stoa explain the disparity (i) by additional files they provided to the Android Device via SDCard, and (ii) running of experiments at their end on a different machine (Intel Xeon(R) CPU @ 3.50GHz, 12 cores, 32GB RAM) with hardware acceleration.

Additionally, we took two measurements to rule out crashes that might be caused by our technique itself (i.e., artificial crashes). First, TimeMachine inserted a delay of 200 ms between any two events to avoid crashes due to Monkey generating many events in a extremely short time. Second, we manually checked stack traces to filter out crashes due to state restoring issues such as inconsistent states.

Finally, our technique tests apps in a virtual machine installed with Android-x86 OS and does not support physical devices yet. For apps interacting with a remote server, our technique saves/restores only app states without considering remote server states. These limitations will be addressed in future work.

7 RELATED WORK

The stream of works most closely related to ours is that of *time-travel debugging* [17, 28, 29, 32, 41]. Time-travel *debugging* allows the user to step back in time, and to change the course of events. The user can now ask questions, such as: "What if this variable had a different value earlier in the execution"? Now, time-travel *testing* has a similar motivation. The tester can test the state-ful app for various, alternative sequences of events, starting in any state.

This work was originally inspired by existing work on *coverage-based greybox fuzzers* (CGF)[2, 18, 19, 39]. A CGF, started with a seed corpus of initial inputs, generates further inputs by fuzzing. If a generated input increases coverage, it is added to the seed corpus. Similar to CGF, our time-travel-enabled test generator maintains a *state* corpus with states that can be restored and fuzzed as needed.

Search-based. The most closely related automatic Android test generation techniques employ search-based methods. Mao et al. developed a multi-objective automated testing technique Sapienz [36]. Sapienz adopts genetic algorithms to optimize randomly generated tests to maximize code coverage while minimizing test lengths. EvoDroid [35], the first search-based framework for Android testing, extracts the interface model and a call graph from app under test and uses this information to guide the computational search process. Search-based approaches are easy to deploy and have attracted a lot of attention from industry, e.g., Sapienz has been used to test different kinds of mobile apps at Facebook. Our work TimeMachine takes a search-based approach as well, but instead of a population of input sequences it evolves a population of app states. Our work proposes a new perspective of app testing as state exploration, and provides a feedback-guided algorithm to efficiently explore an app's state space.

Random. One of the most efficient approaches for testing Android apps is the random generation of event sequences [23]. Apps are exercised by injecting arbitrary or contextual events. Monkey [3] is Google's official testing tool for Android apps, which is built into the Android platforms and widely used by developers. Monkey generates random user events such as clicks, touches, or

gestures, as well as a number of system-level events. Dynodroid [34] employs a feedback-directed random testing approach with two strategies: *BIASEDRANDOM* favors events related to the current context, and *FREQUENCY* is biased towards less frequently used events. Although random testing has gained popularity because of its ease of use, it suffers from an early saturation effect, i.e., it quickly stops making progress, e.g., in terms of coverage. From this point of view, our work powers random testing with the ability to jump to a progressive state observed in the past when there is no progress. Thus, an early saturation can be avoided.

Model-based. Another popular approach of Android apps testing is model-based testing. App event sequences are generated according to models which are manually constructed, or extracted from project artefacts such as source code, XML configuration files and UI runtime state. Ape [26] leverages runtime information to evolve an initial GUI model to achieve more precise models. Stoa [40] assigns widgets in a GUI model with different probabilities of being selected during testing and adjusts them based on feedback such as code coverage to explore uncovered models. AndroidRipper [13] uses a depth-first search over the user interface to build a model. A³E [15] explores apps with two strategies: *Targeted Exploration* which prioritizes exploration of activities that are reachable from the initial activity on a static activity transition graph, and *Depth-first Exploration* which systematically exercises user interface. Droidbot [30], ORBIT [43] and PUMA [27] use static and dynamic analysis to build basic models, on top of which different exploration strategies can be developed. Model-based approaches have attracted a great deal of attention in this field because they allow to represent app behavior as a model on which various exploration strategies can be applied. However, complex widgets (e.g., animation) commonly-used in modern apps pose difficulties on model construction, leading to an incomplete model. Combining model-based approaches with other techniques such as random testing can be a promising option for Android apps testing.

Learning-based. A different line of work uses machine learning to test Android apps. Liu et al. [33] use a model learned from a manually crafted data set (including manual text inputs and associated contexts) to produce text inputs that are *relevant* to the current context during app testing. For instance, it would use a name for an existing city when generating an input for a search box if there is a nearby item labeled Weather. Wuji [44] employs evolutionary algorithms and deep reinforcement learning to explore the state space of a game under test. SwiftHand [20] uses machine learning to learn a model of the user interface, and uses this model to discover unexplored states. The technique by Degott et al. [24] leverages reinforcement learning to identify valid interactions for a GUI element (e.g., a button allows to be clicked but not dragged) and uses this information to guide execution. Humanoid [31] takes manual event sequences and their corresponding UI screens to learn a model and uses the model to predict human-like interactions for an app screen. Machine learning is typically applied to resolve specific challenge in the event sequence generation, such as generating contextual text inputs or identifying possible types of input events that can be executed upon a GUI element. In contrast, our work features a fully automated Android event sequence generator. Some components in TimeMachine such as identifying an interesting state might benefit from machine learning since learning-based

approaches have shown to be effective for similar issues. It is worth to explore this direction in future work.

Program analysis-based. Several existing approaches employ program analysis when testing Android apps. ACTEve [14] uses symbolic execution to compute enabled input events in a given app state and systematically explores the state by triggering these events. SynthesiSE [25] leverages concolic execution and program synthesis to automatically generate models for Android library calls. CrashScope [38] combines static and dynamic analysis to generate an event sequence that is used to reproduce a crash. Thor [12] executes an existing test suite under adverse conditions to discover unexpected app behavior. Such program analysis provides detailed information about the app, which can help to guide test sequence generation. At the same time, intrinsic limitations of program analysis such as poor scalability create an impediment to easy and widely-applicable automation. In contrast, our work TimeMachine requires a little information from app under test and can be easily deployed in practice.

8 CONCLUSION

Android app testing is a well-studied topic. In this paper, we develop time-travel app testing which leverages virtualization technology to enhance testing techniques with the ability to capture snapshots of the system state— state of the app and all of its environment. Capturing snapshots facilitates our time-travel-enabled testing tool to visit arbitrary states discovered earlier that have the potential to trigger new program behavior. State-level feedback allows our technique to accurately identify progressive states and travel to them for maximizing progress in terms of code coverage and state space exploration.

Our time-travel strategy enhances a testing technique (Monkey extended with Stoa's system-level generator) by achieving 1.15 times more statement coverage and discovering 1.7 times more crashes on the AndroTest benchmark. Moreover, TimeMachine outperforms other state-of-the-art techniques (Sapienz and Stoa) by achieving the highest coverage on average and the most found crashes in total. On large, industrial apps, TimeMachine covers around 900 more methods on average and discover 1.5 times more unique crashes over the baseline tool, at the same time outperforms state-of-the-art techniques as well. Our tool TimeMachine also reveals a large number of crashes, owing to a wide variety of exceptions (nine different kinds of exceptions), in real-life top popular apps from Google Play.

ACKNOWLEDGMENTS

This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by NRF Singapore under National Cybersecurity R&D (NCR) programme, and an AcRF Tier1 project T1 251RES1708 from Singapore. This research was partially funded by the Australian Government through an Australian Research Council Discovery Early Career Researcher Award (DE190100046).

REFERENCES

- [1] 2018. 2016 NowSecure Mobile Security Report. (2018). <https://info.nowsecure.com/rs/201-XEW-873/images/2016-NowSecure-mobile-security-report.pdf>
- [2] 2018. American Fuzzy Lop Fuzzer. (2018). <http://lcamtuf.coredump.cx/afl/>

- [3] 2018. Monkey. (2018). <https://developer.android.com/studio/test/monkey>
- [4] 2018. VMWare VirtualBox. (2018). <https://www.virtualbox.org/>
- [5] 2019. Android Debug Bridge. (2019). <https://developer.android.com/studio/command-line/adb>
- [6] 2019. Android-x86. (2019). <http://www.android-x86.org/>
- [7] 2019. Anymemo. (2019). <https://anymemo.org/>
- [8] 2019. ELLA: A Tool for Binary Instrumentation of Android Apps. (2019). <https://github.com/saswatanand/ella>
- [9] 2019. EMMA: a free Java code coverage tool. (2019). <http://emma.sourceforge.net/>
- [10] 2019. Google UI Automator. (2019). <https://developer.android.com/training/testing/ui-automator>
- [11] 2019. A python library for VirtualBox. (2019). <https://pypi.org/project/pyvbox/>
- [12] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Möller. 2015. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12–17, 2015*. ACM, 83–93. <https://doi.org/10.1145/2771783.2771786>
- [13] 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 *IEEE/ACM International Conference on Automated Software Engineering, ASE '12, Essen, Germany, September 3–7, 2012*. ACM, 258–261. <https://doi.org/10.1145/2351676.2351717>
- [14] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, 59:1–59:11.
- [15] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. 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*. ACM, 641–660. <https://doi.org/10.1145/2509136.2509549>
- [16] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 238–249.
- [17] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-travel Debugging for JavaScript/Node.js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, 1003–1007.
- [18] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 1–16.
- [19] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2018. Coverage-based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* (2018), 1–18.
- [20] Wontae Choi, George 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 '13)*, 623–640.
- [21] 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>
- [22] 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 (ICSE '18)*. ACM, New York, NY, USA, 445–455. <https://doi.org/10.1145/3180155.3180173>
- [23] Shaubik 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 '15)*. IEEE Computer Society, Washington, DC, USA, 429–440. <https://doi.org/10.1109/ASE.2015.89>
- [24] Christian Degott, Nataniel P. Borges Jr., and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15–19, 2019*. ACM, 296–306. <https://doi.org/10.1145/3293882.3330569>
- [25] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android Testing via Synthetic Symbolic Execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 419–429. <https://doi.org/10.1145/3238147.3238225>
- [26] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019*. IEEE / ACM, 269–280. <https://doi.org/10.1109/ICSE.2019.00042>
- [27] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14, Bretton Woods, NH, USA, June 16–19, 2014*. ACM, 204–217. <https://doi.org/10.1145/2594368.2594390>
- [28] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-travel Debugging with First-class Traces. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*, 352–361.
- [29] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2005. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*, 1–1.
- [30] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume*. IEEE Computer Society, 23–26. <https://doi.org/10.1109/ICSE-C.2017.8>
- [31] Y. Li, Z. Yang, Y. Guo, and X. 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)*, 1070–1073. <https://doi.org/10.1109/ASE.2019.00104>
- [32] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jin Song Dong. 2017. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017*. IEEE / ACM, 393–403. <https://doi.org/10.1109/ICSE.2017.43>
- [33] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017*. IEEE / ACM, 643–653. <https://doi.org/10.1109/ICSE.2017.65>
- [34] Aravind Machiry, Rohan Tahirani, 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 (ESEC/FSE 2013)*, 224–234.
- [35] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, 599–609.
- [36] 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 (ISSTA 2016)*. ACM, New York, NY, USA, 94–105.
- [37] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, 559–570.
- [38] Kevin Moran, Mario Linares Vázquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11–15, 2016*. IEEE Computer Society, 33–44. <https://doi.org/10.1109/ICST.2016.34>
- [39] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019), 1–17.
- [40] 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 (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.
- [41] Nicolas Viennot, Siddharth Nair, and Jason Nieh. 2013. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, 127–138.
- [42] 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 (ASE 2018)*. ACM, New York, NY, USA, 738–748. <https://doi.org/10.1145/3238147.3240465>
- [43] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE '13)*, 250–265.
- [44] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. 2019. Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 772–784. <https://doi.org/10.1109/ASE.2019.00077>