Time-travel Testing of Android Apps

摘要

安卓测试工具会生成输入事件序列，以此来检测被测应用的状态空间。现有的基于搜索的技术技术系统地演化出事件序列群以达到特定目的：比如最大代码覆盖率等。我们希望通过对合适的事件进行改变，从而生成更合适的序列。但是，事件的改变方向可能是无效的。我们的主要见解是，对原始序列的适应性有贡献的相关应用程序状态可能不会被变异改变的事件序列达到。

在本文中，我们提出了一种演化状态的方法，这些状态可以在发现时被捕获，并在需要时恢复。我们希望在适应的程序状态上生成事件会导致过渡到更适应的状态。例如，我们可以快速取消对主屏幕状态的测试，因为大多数事件序列都访问该状态，而将有限的资源集中在测试更有趣的状态上，否则很难达到。

因为这种方法能够回到过去观察到的任何状态，我们将其称为时间旅行 (Time-Travel) 测试。我们将时间旅行测试实现到 TimeMachine 中，这是一个启用时间旅行的成功的自动化 Android 测试工具 Monkey 的版本。在大量开源和闭源 Android 应用程序的实验中，TimeMachine 在覆盖率和发现的崩溃方面均优于最先进的基于搜索的测试工具 Sapienz 和基于模型的 Android 测试工具 Stoat。

关键词： Android 测试；事件生成；代码覆盖率

1 引言

截至 2020 年，每三个人中就有一部智能手机（29 亿用户），而应用收入将从 2016 年的 880 亿美元增加到 1890 亿美元。移动应用中的错误和漏洞数量正在飞速增加。2016 年，24.7% 的移动应用程序至少包含一个高风险安全漏洞。统计表明，预计 Android 测试市场在五年内将从 2016 年的 32.1 亿美元增加到 2021 年的 63.5 亿美元。在未来，业界对于 Android 测试工具的效率要求将会越来越高。

2 相关工作

在 Android 测试的工作中，生成测试用例的事件序列是一个很关键的步骤，因为这将影响被测软件的覆盖率的水平，同时影响检测出的问题数目大小。下面将介绍测试事件生成领域上的一些普遍方法。

2.1 传统事件生成方法

一种可能的方法是以随机方式生成一个非常长的事件序列 [1]。然而，测试工具最终可能会陷入死胡同。通过重新启动 Android 应用程序，这个问题只能在一定程度上得到解决，因为 (i) 我们必须从头开始，(ii) 没有清空，例如，数据库条目保留，以及 (iii) 如何检测我们何时陷入困境仍然是一个未解之谜。对于 Android 测试来说，能够保存并回溯到最有趣的状态在更系统地探索状态空间方面起到了重要的作用。

2.2 基于搜索的事件生成方法

另一种 Android 应用测试方法 [6] 是以搜索为基础的方式演化事件序列的集合。在每次迭代中，选择适应性最强的事件序列进行变异，以生成下一代事件序列。通过添加、修改或删除任意事件来变异事件序列。然而，这种方法不允许通过从状态遍历各种可用事件来进行系统状态空间的探索。如果在序列 $E = \langle e_1, ..., e_i, ...e_n \rangle$ 中对 e_i 进行变异，那么以 $e_i + 1$ 开头的后缀可能不再可用。

3 本文方法

3.1 本文方法概述

该论文设计了一个通用的 Android 测试时间旅行 (Time Travel) 框架，允许我们在需要时即时保存特定发现的状态并在需要时恢复它。图1展示了时间旅行基础的框架图。Android 应用可以由开发人员自己或自动测试生成器来启动。当与应用进行交互时，状态观察模块将会记录状态转换并监视代码覆盖情况的变化。满足预定义标准的状态被标记为“有趣”，并通过对整个模拟 Android 设备保存系统快照。同时，框架观察应用执行，以确定何时缺乏进展，即测试工具在大量状态转换过程中无法发现任何新的程序行为时。当检测到“缺乏进展”时，框架终止当前执行，选择并恢复先前记录的最可进步的执行状态。更可进步的状态 (more progressive state) 是指能够快速发现更多状态的状态。当我们回到可进步的状态时，会生成一系列新的事件以快速发现新的程序行为。

该框架设计为易于使用且高度可配置。现有的测试技术可以通过实施以下策略部署到该框架中：

- 指定构成“有趣”状态的标准，例如，为了增加代码覆盖率，只有哪些状态将被保存。
- 指定构成“缺乏进展”的标准，例如，当测试技术在循环中遍历相同的状态序列时作为一个时机。
- 提供一个算法，在检测到缺乏进展时选择最可进步的状态进行状态的回溯。

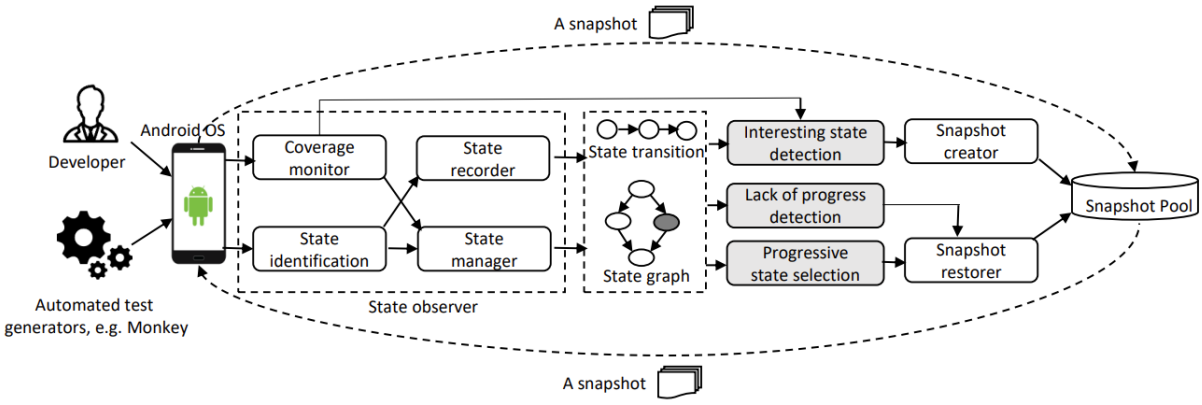

图 1. 时间旅行框架 [5]

TimeMachine 的算法过程如图2所示，其目标是最大化状态和代码的覆盖。该程序从初始状态的快照开始（第 1-4 行）。对于 Monkey [1] 生成的每个事件，计算新状态并更新状态转换图（第 5-9 行）。如果状态是“有趣的”（Sec. 3.1），则获取 VM 的快照并与该状态关联（第 10-13 行）。如果 Monkey 被卡住并且没有更多的进展（Sec. 3.2），TimeMachine 找到最可进步的状态（selectFittestState；Sec. 3.3），并还原关联的 VM 快照（第 14-17 行）。否则，生成一个新事件并开始新的循环（第 5-18 行）。

Algorithm 1: Time-travel testing (TimeMachine).

Input: Android *App*, Sequence generator *Monkey*

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

Output: State Transition Graph *stateGraph*

图 2. 算法一：时间旅行测试 [5]

3.2 定义“有趣”的节点

TimeMachine 基于 GUI 或代码覆盖的变化来识别有趣的状态（Algorithm 1 中的第 10 行）。函数 isInteresting(state) 如果满足以下条件则返回 true：(i) 状态是第一次访问，以及

(ii) 当首次到达状态时执行了新的代码。

对“有趣”状态的定义背后的直觉是，新代码的执行提供了证据，即在发现的状态中启用了以前未测试过的功能。通过探索这个状态，可能会执行与功能相关的更多新代码。例如，假设在屏幕 S1 上点击一个按钮会导致跳转到新的屏幕 S2，从那里显示一个新的组件（增加代码覆盖率）。新的组件带有尚未执行的事件处理程序。通过进一步探索屏幕 S2，可以覆盖这些事件处理程序。这种启发式方法不仅准确地识别了一个“有趣”的状态（在这种情况下是 S2），而且还显著减少了保存状态的总数。

3.3 定义缺乏进展的状态

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

```
Input: Queue length l
Input: Lack-of-progress threshold maxNoProgress
Input: Max. top (α · 100)% most frequently visited states
Input: Max. proportion β of repeated plus frequent states
 1: FIFO Queue ← empty queue of length l
 2: noProgress = 0          // #events since last state transition
 3:
 4: procedure isStuck(State curState, Graph stateGraph) {
 5:   prevStateID = Queue.top()
 6:   if prevStateID == curState.ID then
 7:     noProgress ← noProgress + 1
 8:   else
 9:     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 Queue.length == l then
16:     nRepeated ←countMaxRepeatedStates(Queue)
17:     nFrequent ←countFreqStates(Queue, stateGraph, α)
18:     if (nRepeated + nFrequent)/l > β then
19:       return true                        // detect loops
20:     end if
21:   end if
22:   return false
23: }
```

图 3. 算法二：检测死循环和死路 [5]

当 TimeMachine 陷入困境时，会回溯到最可进步的状态（图3算法二中的第 14-17 行）。函数 isStuck 在图2中进行了概述，并实现了一个滑动窗口算法。首先，必须指定四个参数：有两个全局变量，一个指定长度的队列 l 和一个计数器，用于追踪相同状态观察到的次数（第 1-3 行）。在给定当前应用程序状态和状态转换图的情况下，如果当前状态与先前状态相同，则无进展计数器会增加（第 4-7 行）。否则，计数器将被重置（第 8-11 行）。如果计数器超过指定的最大值（maxNoProgress），则检测到一个死胡同（第 12-14 行）。如果固定长度的队列已满，并且队列中“简单”状态的比例超过指定的阈值 β，则说明检测到一个循环。队列中被视

为“简单”的两种状态是：在队列中多次发生的状态，以及在最常访问的状态的前 α 百分比之内的状态。

3.4 可进步的状态的选择

每当 Monkey 被卡住，为了选择一个状态回溯到，算法为每个状态都分配了一个适合度 (fitness)，评估它触发新程序行为的潜力（图2中的第 14-17 行）。状态 s 的适合度 $f(s)$ 由该状态被访问的次数和从中生成的“有趣”状态的次数确定。具体而言，适合度函数定义如下：

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

其中：

- $v(s)$ 是状态 s 被访问的次数
- $w(s)$ 是从状态 s 生成的“有趣”状态的次数
- r 是找到有趣状态的奖励常数
- p 是过渡到已经被发现的状态的惩罚常数
- f_0 是初始值

在 TimeMachine 中，有趣状态的初始值被设置为不有趣状态的 6 倍，而 r 和 p 都被设置为 0.1。当一个状态被反复访问且没有发现有趣的状态时，由于惩罚 p，其适应度将不断减小，以便最终选择并恢复其他状态。

Algorithm 3: Selecting the next state (SELECTFITTESTSTATE)

Input: Path length k

```
1:  procedure SELECTFITTESTSTATE(states, stateGraph) {
2:    bestFitness ← 0
3:    for each state in states do
4:      stateFitness ← 0
5:      paths ← 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 ← stateFitness + f(s) // see Eq. (1)
9:        end for
10:     end for
11:     stateFitness ← stateFitness / |paths|
12:     if stateFitness > bestFitness then
13:       bestState = state
14:       bestFitness = stateFitness
15:     end if
16:   end for
17:   return bestState
18: }
```

图 4. 算法三：选择下一个状态 [5]

图4算法三概述了选择最可进步的状态进行时间旅行的过程。它的输入是具有关联的 VM 快照中的“有趣”状态，以及由时间旅行框架维护的状态转换图。确定状态的“邻域”的转换数 k 必须由用户指定，而在实验中，默认设置 $k = 3$。对于每个有趣的状态，TimeMachine 计算一个状态在 k 邻域范围内的平均适合度。

4 复现细节

4.1 与已有开源代码对比

源代码位于 Github 仓库：https://github.com/DroidTest/TimeMachine 中，该仓库实现了所有的时间旅行（Time Travel）框架细节。

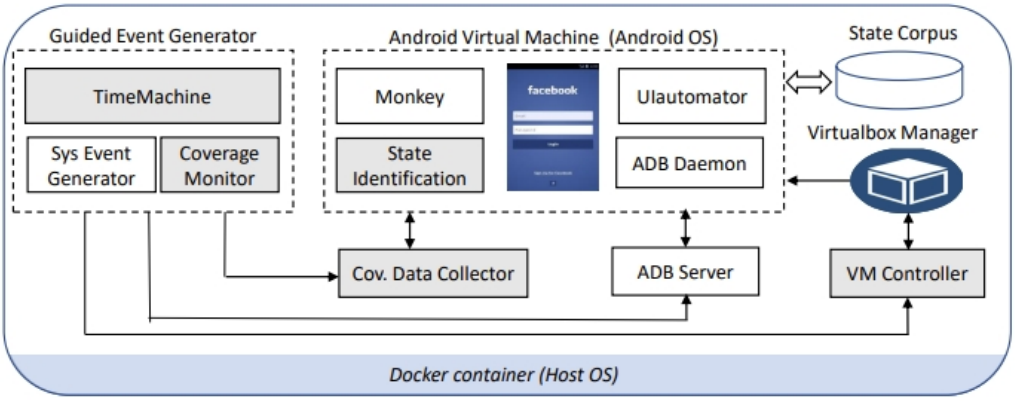

图 5. 时间旅行的实现架构 [5]

时间旅行框架被实现为一个完全自动化的应用程序测试平台，使用或扩展以下的工具：VirtualBox，用于运行和控制 Android-x86 OS 的 Python 库 pyvbox，用于观察状态转换的 Android UI Automator，以及与正在测试的应用程序交互的 Android Debug Bridge (ADB) [2]。图5提供了该实现的平台级体系结构概述。灰色的组件由论文工作者实现，而其他组件是使用或修改现有的工具。

对于覆盖率收集，实现框架使用 Emma [4]（状态覆盖率）对开源应用进行测试操作，对闭源应用使用 Ella [3]（方法覆盖率）。Ella 使用一个客户端-服务器模型，通过套接字连接将覆盖数据从 Android OS 发送到 VM 主机。需要注意的是，每次还原快照时，此连接都会中断。为解决这个问题，框架源码作者修改了 Ella，将覆盖数据保存在 Android OS 上，以便根据需要主动提取。

4.2 实验环境搭建

准备以下环境：

- Ububntu 18.04 64-bit 或 Mac-OSX 10.15
- Android SDK with API 25 (确保 adb, aapt, avdmanager, emulator 都已正确配置)
- Python 2.7 (确保 enum 和 uiautomator 已经完成安装)

首先需要完成 Jacoco 的配置：将 Jacoco 的测试操作代码注入到待测 Android 软件中，使用 gradlew 构建工具完成 Android 软件的 Debug 版本的构建。再开启 android 模拟器，使用 adb 将构建好的 app 安装到模拟器当中的 android 系统中。最终开启自动化脚本调用 TimeMachine 测试代码开始自动化测试。

4.3 工具使用和工作流程说明

当使用 TimeMachine 开始自动化测试时，事件生成工具会不断产生事件序列，并将产生的事件映射到模拟器中的 Android 目标测试软件上，事件由模拟器的控制层完成渲染生成对应的测试用例。在事件生成的同时，TimeMachine 会不断记录由事件的触发而产生的系统状态，并将这些状态有选择地记录在状态池中。最终完成的效果就是下图6所示，在右边是 Android 模拟器，左边是在测试的事件所产生的日志信息。

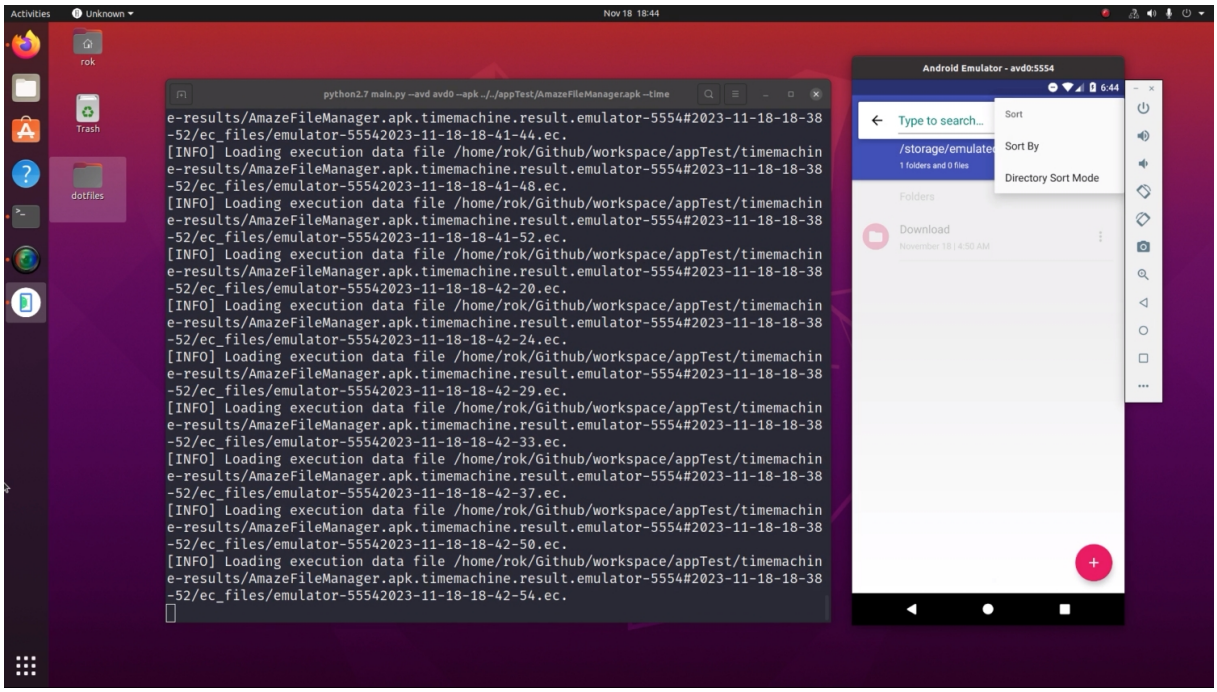

图 6. 测试软件操作示意 [5]

5 实验结果分析

图7显示了每种技术在 68 个 (部分)Android 应用程序上实现的覆盖率和发现的故障。每个应用程序的最高覆盖率和发现的崩溃都以灰色突出显示。TimeMachine 和基准技术的结果显示在 “TimeMachine” 和 “Baselines” 列中。需要注意的是，MS 表示使用了 Stoat 的系统级事件生成器的 Monkey，而 MR 表示 Monkey 具有在检测到缺乏进展时从头重新启动测试的能力。

Subjects	TimeMachine			Baselines				State-of-the-art					
	%Cov	#Cra	#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
baterrydog	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

图 7. 对比测试结果（在开源软件下）[5]

TimeMachine 与 MS 的比较分析。TimeMachine 平均达到了 54% 的语句覆盖率，并检测到 68 个基准应用程序的 199 个独特崩溃。MS 平均达到了 47% 的语句覆盖率，并检测到 115 个独特的崩溃。TimeMachine 涵盖了 1.15 倍的语句，并比 MS 多揭示了 1.73 倍的崩溃。可以看到对于所有的被测试的应用程序大小组而言，覆盖率的提高都是实质性的。

TimeMachine 与 MR 的比较分析。MR 平均实现了 47% 的语句覆盖率，并在 68 个基准应用程序中检测到 45 个独特的崩溃。TimeMachine 实现了比 MR 高 1.15 倍的语句覆盖率和 4.4 倍的独特崩溃。同样，图7显示 TimeMachine 在短时间内覆盖了更多的代码，并且与 MR 相比，对于所有四个应用程序大小组，它在实现语句覆盖率方面都有了实质性的改进。这表明仅仅在检测到缺乏进展时从头重新启动应用程序是不足够的，尽管 MR 通过提高 Monkey 的 3% 语句覆盖率。

图8显示了 37 个闭源基准应用程序的测试结果。很明显，TimeMachine 在所有评估技术中实现了最高的方法覆盖率 19% 和最多的崩溃 281。与基准 MS 和 MR 相比，TimeMachine 分别将方法覆盖率从 17% 和 15% 提高到了 19%。需要注意的是，每个应用程序平均有 44182 个方法，因此对于每个应用程序，覆盖的方法数量增加了大约 900 到 1800 个，这 2% 到 4%

的提高是相当可观的。就发现的崩溃数量而言，TimeMachine 分别比 MS 和 MR 多发现了 1.5 倍和 18.7 倍的崩溃。MS 检测到 183 个崩溃，MR 检测到 15 个崩溃。

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

图 8. 对比测试结果（在闭源软件下）[5]

6 总结与展望

Android 应用测试是一个经过深入研究的课题。在本文中，工作者开发了一种利用虚拟化技术的时间旅行应用测试，以增强测试的性能，该技术具有捕获系统状态快照的能力——状态包含被测应用及其所有环境的状态。捕获快照使得支持时间旅行的测试工具能够访问先

前发现的任意状态，这些状态有可能触发新的程序行为。状态级反馈使文中的技术能够准确识别渐进状态，并在代码覆盖和状态空间探索方面最大限度地实现进展。

时间旅行策略实现了对 AndroTest 基准测试的 1.15 倍状态覆盖和发现多达 1.7 倍崩溃的优势，该性能水平数据说明该技术能够有效增强测试技术（使用带有 Stoat 系统级生成器的 Monkey 进行扩展）。此外，TimeMachine 在平均值上表现优异，击败了其他最先进的技术（Sapienz 和 Stoat），实现了最高的覆盖率和总体上发现最多的崩溃。在大型工业应用上，TimeMachine 平均涵盖了约 900 个额外的方法，并在相同时间内优于其他最先进的技术，发现的未触发的崩溃次数增加了 1.5 倍。实现的工具 TimeMachine 还在来自 Google Play 的真实热门应用中测试出了大量崩溃，并发现了各种不同的异常（九种不同类型的异常）是造成这些崩溃的原因。

参考文献

[1] *Monkey*, 2018. https://developer.android.com/studio/test/monkey.

[2] *Android Debug Bridge*, 2019. https://developer.android.com/studio/command-line/adb.

[3] *ELLA: A Tool for Binary Instrumentation of Android Apps*, 2019. https://github.com/saswatanand/ella.

[4] *EMMA: a free Java code coverage tool*, 2019. http://emma.sourceforge.net/.

[5] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 481–492, 2020.

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