An Empirical Study of Android Test Generation Tools in Industrial Cases

Wenyu Wang
University of Illinois at Urbana-Champaign, USA
wenyu2@illinois.edu

Dengfeng Li
University of Illinois at Urbana-Champaign, USA
dli46@illinois.edu

Wei Yang
University of Texas at Dallas, USA
weiyang.utd@gmail.com

Yurui Cao
University of Illinois at Urbana-Champaign, USA
yuruic2@illinois.edu

Zhenwen Zhang
Yuetang Deng
adazhang@tencent.com
yuetangdeng@tencent.com
Tencent Inc., China

Tao Xie
University of Illinois at Urbana-Champaign, USA
taoxie@illinois.edu

ABSTRACT

User Interface (UI) testing is a popular approach to ensure the quality of mobile apps. Numerous test generation tools have been developed to support UI testing on mobile apps, especially for Android apps. Previous work evaluates and compares different test generation tools using only relatively simple open-source apps, while real-world industrial apps tend to have more complex functionalities and implementations. There is no direct comparison among test generation tools with regard to effectiveness and ease-of-use on these industrial apps. To address such limitation, we study existing state-of-the-art or state-of-the-practice test generation tools on 68 widely-used industrial apps. We directly compare the tools with regard to code coverage and fault-detection ability. According to our results, Monkey, a state-of-the-practice tool from Google, achieves the highest method coverage on 22 of 41 apps whose method coverage data can be obtained. Of all 68 apps under study, Monkey also achieves the highest activity coverage on 35 apps, while Stoat, a state-of-the-art tool, is able to trigger the highest number of unique crashes on 23 apps. By analyzing the experimental results, we provide suggestions for combining different test generation tools to achieve better performance. We also report our experience in applying these tools to industrial apps under study. Our study results give insights on how Android UI test generation tools could be improved to better handle complex industrial apps.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Android UI testing, test generation, empirical study

ACM Reference Format:
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 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3240465

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240465

1 INTRODUCTION

As the fast pace of Android app development and evolution continues, effective quality assurance for industrial Android apps becomes increasingly necessary and demanding. User Interface (UI) testing, aiming to uncover potential app defects (*e.g.*, crashing) by mimicking human interactions, has long been an important approach to ensure the quality of Android apps before their delivery to end users. To facilitate UI test automation, the Android developer toolkit from Google provides Monkey [17], an automatic test generation tool that sends randomly generated UI event sequences to an app under test. In addition, researchers have also proposed various test generation tools to automate Android UI testing [1, 3, 6, 7, 23, 24, 26, 27, 29, 31–33].

These industrial or academic test generation tools all show satisfactory performance according to their own respective evaluation on various open-source or industrial apps. Table 1 shows the statistics of subjects used for evaluating existing Android test generation tools (published in major software engineering conferences). The last row of the table also shows a study conducted by Choudhary et al. [8] in 2015 by comparing different Android test generation tools. In the table, coverage comparison (denoted as 'Emp. Comp.') shows the numbers of open-source apps (denoted as '#Opn.') and industrial apps (denoted as '#Ind.') used in evaluating the proposed tool's capability in terms of code coverage or/and fault detection (against other tools). By default, the code coverage is compared across tools. We mark with ♣ these entries where both code coverage and fault detection are compared. 'Case #Ind.' shows the numbers of industrial apps used in case studies for the proposed tools. These case studies does **not** report code coverage or compare the proposed tool against other related previous tools. The sole purpose of these

Table 1: Overview of existing Android test generation tools and their evaluation subjects

Tool/Study	Venue	Emp. Comp.		Case
		#Ind.	#Opn.	#Ind.
A^{3}E [6]	OOPSLA'13	0	0	◇28
ACTEve [3]	FSE'12	0	5	0
DroidBot [24]	ICSE-C'17	0	2	0
Dynodroid [26]	FSE'13	0	50	1000
GUIRipper [1]	ASE'12	0	♣1	0
Monkey [17]	-	-	-	-
Sapienz [27]	ISSTA'16	0	♣68	1000
Stoat [29]	FSE'17	0	♣93	1661
SwiftHand [7]	OOPSLA'13	0	10	0
WCTester [32]	FSE-Ind'16	△1	0	0
Study by [8]	ASE'15	0	♣68	0

♣: Both code coverage and fault detection are compared across tools on open source apps.
◇: Only code coverage is measured whereas fault detection is not measured on industrial apps.
△: The tool is compared with only Monkey but no other tools.

studies is to evaluate the proposed tools' applicability on industrial Android testing tasks (by reporting the results of **only** fault detection). One exception there is A^{3}E [6], which is evaluated on only code coverage without on fault detection (note that no tool comparison is conducted there).

As shown in Table 1, there exists **no comparison among existing tools over industrial apps in terms of both code coverage and fault detection**. Subjects for empirical tool comparison (in the 'Emp. Comp.' column) include only open-source apps, with one exception (WCTester [32]) where the proposed tool is compared with only **one** baseline tool (Monkey) on only **one** industrial app (WeChat). In addition, although the case-study evaluation of a few tools includes industrial apps (in the '#Case Ind.' column), no tool comparison is conducted there, and no case studies on industrial apps measure both code coverage and fault detection. There exist a gap and yet a strong need to investigate and compare how well these proposed tools perform on industrial apps that, in contrast to open-source apps, are usually (1) much more complex with regard to functionalities and implementations, (2) better maintained and tested, and (3) with much larger user bases and higher impacts.

To fill this gap and give practitioners and researchers insights on how existing tools perform on industrial apps, in this paper, we present the first empirical study that conducts comparison among existing tools on industrial apps in terms of both code coverage and fault detection. In particular, we investigate how existing available state-of-the-art or state-of-the-practice test generation tools perform on 68 widely-used industrial apps in terms of code coverage (method and activity coverage) and fault detection (the number of distinct triggered crashes). These apps span across 30 different categories and each of these apps has at least 1 million installs according to Google Play [15]. We empirically study the coverage and fault-detection results to gain insights on each tool's strengths and weaknesses. We also study how to efficiently combine some of these tools to achieve better code coverage or fault detection capabilities on testing industrial apps. We also report our experience in applying these tools to testing tasks for industrial Android apps.

In this paper, our empirical study provides app developers and tool researchers/vendors with insights on the strengths and weaknesses of existing test generation tools, helping them improve their tools' design and implementation and their handling of realistic tasks for industrial apps. In particular, we address four main research questions in our study:

- **RQ1:** What is the code coverage (method coverage and activity coverage) achieved by each test generation tool under study on applicable industrial apps?
- **RQ2:** How many unique crashes can each test generation tool trigger on each applicable industrial app? What are the causes of these crashes?
- **RQ3:** How to efficiently combine multiple test generation tools on applicable industrial apps to achieve better coverage and fault detection than applying these tools individually?
- **RQ4:** How much effort does it require to set up each test generation tool for testing industrial apps?

We run different test generation tools under study on selected industrial apps to study the effectiveness of these tools. According to our results, Monkey achieves the highest method coverage on 22 of 41 apps whose method coverage data can be obtained. Of all 68 apps under study, Monkey also achieves the highest activity coverage on 35 apps, while Stoat is able to trigger the highest number of unique crashes on 23 apps.

To gain better understanding of the tool performance on industrial apps, similar to a previous study methodology [21], we rank each covered method/activity or triggered unique crash in each industrial app based on the number of test generation tools that have covered the method/activity or triggered the unique crash. For instance, a method/activity or unique crash is considered rank-1 if only one test generation tool has covered the method/activity or triggered the unique crash. Our results show that, on many industrial apps, Monkey has the highest numbers of rank-1 methods and activities, and Stoat is able to trigger the highest numbers of rank-1 unique crashes. Our analysis also provides suggestions for combining multiple tools for better coverage and/or fault detection than applying these tools individually.

In summary, this paper makes the following main contributions:

- Empirical investigation on the effectiveness of existing available Android UI test generation tools when being applied on industrial apps.
- Detailed analysis of the coverage results achieved by each tool to provide insights on the strengths and weaknesses of the test generation tools under study and on how to better leverage these tools.
- Hands-on experience report of applying multiple state-of-the-art or state-of-the-practice test generation tools on complex industrial apps.
- A strong implication that testing researchers for Android test generation tools should empirically compare a newly proposed tool with related previous tools on industrial apps ***besides*** open-source apps, going *beyond* the current common research practice of comparing tools on *only* open-source apps.

2 BACKGROUND

In this section, we present an overview of the Android app components and the Android OS architecture.

2.1 Android App Components

From the view of users and other apps, an Android app consists of four types of components: Activities, Intent Filters, Services, and Content Providers [14].

Activities. Activities are designed to show UI screens consisting of sets of layouts and UI widgets (*e.g.*, buttons). Widgets are associated with sets of attributes (*e.g.*, sizes and positions) and can be bound to callback methods to handle UI events (*e.g.*, short clicks). An activity is typically used for a single specific scenario such as logging in and user registration.

Intent Filters. Intents are messaging objects used by components within an app or across different apps to communicate with each other. Intent filters are used to allow only the designated intent types to be received and processed by the components. Launching an app, for instance, is achieved by sending a specific intent to the app main activity that intercepts such intents.

Services. Services are intended to perform tasks in the background without being attached to a UI screen. Downloading tasks, for example, are usually implemented using services to avoid blocking the usage of app main functionalities.

Content Providers. Content providers enable apps to expose and manage a globally shared set of data. For instance, a user's contact information is stored in an Android system app and may be accessed by other apps using the specific content provider.

2.2 Android OS Architecture

The Android OS is an open-source Linux-based software stack [18]. Android apps run within individual sandboxes, namely instances of the Dalvik Virtual Machine (DalvikVM) [12], on top of native libraries and the Linux kernel. Android frameworks, which are responsible for low-level functionalities of the Android OS including UI and activity management, also run within instances of the DalvikVM and can be reached by apps via Android APIs. System apps are pre-installed on the Android OS to provide users with basic features such as phone calling and SMS sending.

The Java source code of an Android app is compiled into dex-code [11] and subsequently packaged as an Android Package (APK) file along with other resource files. Developers are also allowed to write their app libraries in C/C++ as native libraries and invoke them through the Java Native Interface (JNI). The app can then be installed on a compatible Android OS. At runtime, the system's DalvikVM reads the app's dex-code and executes it. Starting from Android 4.4, the Android Runtime (ART) is included with the Android OS, where the ART translates and optimizes dex-code to native machine code during installation to enable faster execution. Note that both DalvikVM and ART have the *64K reference limitation* (*i.e.*, there cannot be more than 65,536 methods in a single `.dex` file that contains an app's dex-code) due to the design of the DalvikVM instruction set. Android provides *multidex* support [13] to mitigate this limitation.

3 SELECTION OF ANDROID TEST GENERATION TOOLS

We choose 6 state-of-the-art or state-of-the-practice UI test generation tools for our study. Monkey [17] is the official test generation tool shipped with all Android devices, while the rest are all published at top venues of software engineering. We select tools that are applicable on at least half of the industrial apps under study. Table 2 presents an overview of the test generation tools that we examine and our decision on tool selection.

In this section, we first present the selected tools for our study. We then illustrate the excluded tools along with the reasons why these tools cannot be included for the study.

3.1 Selected Tools under Study

3.1.1 Monkey. Monkey [17] is a purely randomized Android test generation tool (from Google) that generates pseudo-random streams of UI events (*e.g.*, clicks, touches, and gestures on UI) and limited types of system-level events (such as volume controls) to unmodified Android apps. Monkey is the most widely used tool in industrial settings due to its applicability to a variety of application settings (*e.g.*, ease of use and compatibility with different Android platforms) [8].

3.1.2 WCTester. To inherit the advantages of Monkey while addressing its major limitations, the WeChat team develops a new approach [32, 33] incorporating three main strategies. First, WCTester finds and triggers only enabled events on each UI screen. Second, WCTester focuses on generating events with higher chances to change current UI states. Third, WCTester considers the UI state history and avoids repetitions during exploration. The new approach leads to significant performance improvements on testing the WeChat app, one of the most popular messenger apps in the world with over 1 billion monthly active users [22].

3.1.3 Sapienz. Sapienz [27, 28] is an evolutionary-testing-based test generation tool for Android UI testing. It leverages a genetic algorithm [9] to evolve generated seed input sequences to search for the optimized test suites containing short input sequences while maximizing code coverage and fault revelation. Pre-defined input sequences (*i.e.*, motif genes) are leveraged to complement the random exploration and provide local exercise for different types of UI widgets. String resources inside apps are extracted as seeds for text inputs. Multi-level instrumentation is supported to accommodate various apps. Test suites can be evaluated simultaneously on multiple devices to speed up the search process.

3.1.4 Stoat. Stoat [29] is a UI test generation tool for Android apps, with model-based evolutionary testing. It constructs a probabilistic UI state-transition model through dynamic exploration and optional static analysis at the first stage. It then evolves the model to search for the optimized model with regard to comprehensive fitness scores of the concrete input sequences derived from Gibbs sampling [4] on models. Code coverage, model coverage, and test-suite diversity are reflected in the fitness score. System-level events are also randomly injected to further enhance the testing effectiveness.

Table 2: Overview of Android test generation tools under study

Tool	Open Source	No Need of Modification		Exploration Strategy	No Need of App Source Code
		App	Platform		
Monkey [17]	✓	✓	✓	Random	✓
WCTester [32, 33]	✗	✓	✓	Random	✓
Sapienz [27]	✓	✓*	✗	Evolutionary	✓
Stoat [29]	✓	✓*	✓	Model-based Evolutionary	✓
DroidBot [24]	✓	✓	✓	Model-based	✓
A^3E-Depth-First [6]	✓	✗	✓	Systematic	✓

* Instrumentation is optional for Sapienz and Stoat.

3.1.5 DroidBot. DroidBot [24] is a programmable, light-weight, and model-based Android UI testing tool. It generates UI-guided test inputs based on a state-transition model constructed on the fly. It also allows developers to write testing scripts to customize the exploration strategy. Detailed testing reports are provided after each test to help developers understand apps' behavior. DroidBot has received over 300 stars on GitHub [25] at the time of writing.

3.1.6 A^3E-Depth-First. A^3E [6] includes a systematic testing tool (*i.e.*, A^3E-Depth-First) that performs a depth-first search strategy during exploration. Such a search strategy mimics user actions and aims to thoroughly cover app functionalities. Another strategy named *Targeted Exploration* is also proposed for fast, direct exploration of activities (as opposed to the general-purpose exploration that aims for higher code coverage or fault detection) in A^3E. The strategy is based on high-level control flow graphs capturing activity transitions and constructed by performing static dataflow analysis on apps' bytecode.

3.2 Excluded Tools and Reasons

This section describes the Android test generation tools that are published in top venues but are not included in our study. We further provide reasons why these Android test generation tools are not applicable for the study.

3.2.1 Dynodroid. Dynodroid [26] is a guided random testing tool that generates user UI events and system-level events. By instrumenting the Android OS, Dynodroid computes the set of relevant events that can execute code of the app under test. Furthermore, Dynodroid generates more system-level events than Monkey such as incoming phone calls and geolocation changes.

Reason. Dynodroid works on only emulators with Android OS version 2.3 due to the requirement of instrumenting the Android platform, and the authors of Dynodroid publish only the instrumented version for Android 2.3. Very few industrial apps under our study still support such an outdated Android system that was released in 2010.

3.2.2 GUIRipper. GUIRipper [1] is a model-based testing tool. It constructs a finite-state-machine (FSM) model of the UI and performs the depth-first search (DFS) exploration strategy. To build the model, GUIRipper instruments the APK file of the app under test and dynamically analyzes the app UI to obtain relevant events related to UI widgets. The tool then systematically traverses the app UI, generating and executing obtained relevant events when new states are encountered.

Reason. We fail to adapt GUIRipper to real devices (note that only a binary version of GUIRipper for the Windows OS is available). In addition, even on emulators, GUIRipper works on only Android 4.0 and it fails to process most industrial apps under study.

3.2.3 SwiftHand. SwiftHand [7] is a model-based testing tool. It features a specialized active learning algorithm to approximate a model of the app under test to guide exploration into unexplored parts of the app's state space. Unlike traditional active learning algorithms such as $\mathcal{L}^*$ [5], such design minimizes the number of restarts during exploration. SwiftHand requires to instrument the APK file of the app under test to obtain the app UI information during testing.

Reason. Due to possible implementation defects, SwiftHand fails on most of the industrial apps under study during instrumentation with error messages such as `ArrayIndexOutOfBoundsException`, or simply never finishes instrumentation and produces GiB-sized log files.

3.2.4 ACTEve. ACTEve [3] is a concolic-testing [10] tool for Android apps. By instrumenting both the Android SDK and the app under test, ACTEve symbolically tracks events from the originating points (*e.g.*, tap coordinates on screen) to the code handling the events. Such approach limits the search space for feasible events and avoids generating redundant inputs. The tool also identifies read-only or ineffective events to further reduce the sizes of event sequences.

Reason. Similar to Dynodroid, ACTEve works on only Android 2.3 and it requires to instrument both the Android SDK and Android apps.

4 STUDY METHODOLOGY

In this section, we present our study methodology including the industrial-app selection, coverage/crash measurement, and study setup.

4.1 Industrial-App Selection

We choose to obtain industrial apps from Google Play, the official Android app market by Google with huge user base. We sample multiple top-recommended apps with the highest numbers of downloads from each category, and manage to harvest 68 industrial apps that are compatible with Android 4.4, the most recent version of Android supported by most of the top-recommended apps and all test generation tools under study. Note that the WeChat app is specifically excluded due to the fact that WCTester, one of the tools

Table 3: Overview of industrial apps under study and their applicability on selected test generation tools

App Name	Version	Category	#Install	Login	#Method	#Activity	Applicability					
							M.	W.	Sa.	St.	D.	A.
Abs	4.2.0	Health & Fitness	10m+	✗	47217	31	✓	✓	✓	✓	✓	✗
AccuWeather	5.3.5-free	Weather	50m+	✗	59429	43	✓	✓	✓	✓	✓	✓
Adobe Acrobat	18.1.0	Productivity	100m+	✗	-	42	✓	✓	✓	✓	✓	✓
Amazon Kindle	8.5.0.77	Books & Reference	100m+	✓	-	123	✓	✓	✓	✓	✓	✓
Autolist	5.2.2	Auto & Vehicles	1m+	✗	-	30	✓	✓	✓	✓	✓	✓
AutoScout24	9.3.14	Auto & Vehicles	10m+	✓	104043	40	✓	✓	✓	✓	✓	✓
Best Hairstyles	1.17	Beauty	1m+	✗	5703	4	✓	✓	✓	✓	✓	✓
CNN	5.1	News & Magazines	10m+	✗	52677	31	✓	✓	✓	✓	✓	✓
Crackle	5.2.1	Entertainment	10m+	✗	52393	16	✓	✓	✓	✓	✓	✓
Duolingo	3.75.1	Education	100m+	✗	60702	55	✓	✓	✓	✓	✓	✓
ES File Explorer	4.1.6	Productivity	100m+	✓	96067	105	✓	✓	✓	✓	✓	✓
Evernote	7.12	Productivity	100m+	✓	89512	160	✓	✓	✓	✓	✓	✓
Excel	16.0.9126	Productivity	100m+	✗	84138	27	✓	✓	✓	✓	✓	✗
Facebook	164.0.0	Social	1b+	✓	-	587	✓	✓	✓	✓	✗	✗
Filters For Selfie	1.0.0	Beauty	1m+	✗	2883	8	✓	✓	✓	✓	✓	✓
Flipboard	4.1.1	News & Magazines	500m+	✓	27527	74	✓	✓	✓	✓	✓	✗
Floor Plan Creator	3.2	Art & Design	5m+	✗	8847	13	✓	✓	✓	✓	✓	✓
Fox News	3.0.0	News & Magazines	10m+	✗	-	31	✓	✓	✓	✓	✓	✓
G.P. Books	4.0.47	Books & Reference	1b+	✗	-	33	✓	✓	✓	✓	✓	✓
G.P. Music	8.7.6773	Music & Audio	1b+	✗	23713	65	✓	✓	✓	✓	✓	✓
G.P. Newsstand	4.7.0	News & Magazines	1b+	✗	70514	32	✓	✓	✓	✓	✓	✓
Gmail	8.3.12	Communication	1b+	✗	-	60	✓	✓	✓	✓	✓	✗
GO Launcher Z	2.51	Personalization	100m+	✗	170699	182	✓	✓	✓	✓	✓	✓
GoodRx	5.3.6	Medical	1m+	✗	50105	61	✓	✓	✓	✓	✓	✓
Google	7.24.32	Tools	1b+	✗	-	117	✓	✓	✓	✓	✓	✗
Google Calendar	5.8.24	Business	500m+	✓	-	32	✓	✓	✓	✓	✓	✗
Google Chrome	65.0.3325	Communication	1b+	✗	-	84	✓	✓	✓	✓	✓	✗
ibisPaint X	5.1.5	Art & Design	10m+	✗	28106	36	✓	✓	✓	✓	✓	✓
Instagram	38.0.0	Social	1b+	✓	-	40	✓	✓	✓	✓	✓	✗
inStar	0.9.8	Art & Design	5m+	✗	52911	23	✓	✓	✓	✓	✓	✓
LINE Camera	14.2.4	Photography	100m+	✗	83214	64	✓	✓	✓	✓	✓	✗
Marvel Comics	3.10.3	Comics	5m+	✗	25563	44	✓	✓	✓	✓	✓	✓
Match	18.03.01	Dating	10m+	✗	52519	66	✓	✓	✓	✓	✓	✓
McDonald	5.12.0	Food & Drink	10m+	✓	-	62	✓	✓	✓	✓	✓	✓
Merriam-Webster	4.1.2	Books & Reference	10m+	✗	25554	17	✓	✓	✓	✓	✓	✓
Messenger	160.0.0	Communication	1b+	✓	-	310	✓	✓	✓	✓	✓	✗
Mirror	30	Beauty	1m+	✗	7215	12	✓	✓	✓	✓	✓	✓
My baby Piano	2.22.2614	Parenting	5m+	✗	726	3	✓	✓	✓	✓	✓	✓
NFL	14.3.46	Sports	50m+	✗	-	46	✓	✓	✓	✓	✓	✓
Nike Run Club	2.14.1	Health & Fitness	10m+	✓	-	113	✓	✓	✓	✓	✓	✗
NOOK	4.7.0.39	Books & Reference	10m+	✗	91032	132	✓	✓	✓	✓	✓	✓
OfficeSuite	9.3.11997	Business	100m+	✗	-	126	✓	✓	✓	✓	✓	✓
OneNote	16.0.9126	Business	100m+	✓	-	76	✓	✓	✓	✓	✓	✗
Photos	3.18.0	Photography	1b+	✗	-	114	✓	✓	✓	✓	✓	✗
Pinterest	6.59.0	Lifestyle	100m+	✓	100420	33	✓	✓	✓	✓	✓	✓
Quizlet	3.15.2	Education	10m+	✓	71511	58	✓	✓	✓	✓	✓	✓
realtor.com	8.13.2	House & Home	10m+	✗	44723	34	✓	✓	✓	✓	✓	✓
Sing!	5.4.1	Music & Audio	100m+	✓	-	53	✓	✓	✓	✓	✓	✗
Sketch	8.0.A.0.2	Art & Design	50m+	✗	-	46	✓	✓	✓	✓	✓	✓
Speedometer	3.6	Auto & Vehicles	1m+	✗	17030	11	✓	✓	✓	✓	✓	✓
Spotify	8.4.48	Music & Audio	100m+	✓	206474	113	✓	✓	✓	✓	✓	✗
TED	3.1.16	Education	10m+	✗	-	27	✓	✓	✓	✓	✓	✓
The Weather Chnl.	8.10.0	Weather	50m+	✗	-	99	✓	✓	✓	✓	✓	✓
Ticketmaster	1.11.0	Events	5m+	✗	-	121	✓	✓	✓	✓	✓	✗
Translate	5.18.0	Tools	500m+	✗	29666	33	✓	✓	✓	✓	✓	✓
TripAdvisor	25.6.1	Food & Drink	100m+	✓	106519	213	✓	✓	✓	✓	✓	✓
trivago	4.9.4	Travel & Local	10m+	✗	34790	29	✓	✓	✓	✓	✓	✓
UC Browser	11.5.0	Communication	500m+	✗	-	63	✓	✓	✓	✓	✓	✓
WatchESPN	2.5.1	Sports	10m+	✗	22686	16	✓	✓	✓	✓	✓	✓
Wattpad	6.82.0	Books & Reference	100m+	✓	89639	93	✓	✓	✓	✓	✓	✓
Waze	4.36.0.1	Maps & Navigation	100m+	✗	-	203	✓	✓	✓	✓	✓	✓
WEBTOON	2.0.4	Comics	10m+	✓	81503	62	✓	✓	✓	✓	✓	✗
Wish	4.16.5	Shopping	100m+	✓	31512	74	✓	✓	✓	✓	✓	✗
Word	16.0.9126	Productivity	100m+	✗	77895	27	✓	✓	✓	✓	✓	✗
Yelp	9.33.0	Food & Drink	10m+	✓	204308	277	✓	✓	✓	✓	✓	✓
YouTube	13.12.60	Video Player & Editor	1b+	✗	-	48	✓	✓	✓	✓	✓	✗
Zedge	5.38.7	Personalization	100m+	✗	138309	35	✓	✓	✓	✓	✓	✓
Zillow	9.4.2	House & Home	10m+	✗	-	82	✓	✓	✓	✓	✓	✓

under study, is specifically optimized for the app and could potentially cause bias in the result. We also manually register accounts for apps that require logging in to access their major functionalities. In addition, apps requiring special/sensitive information (*e.g.*, banking) or related to real-world services (*e.g.*, taxi calling) are skipped to minimize undesirable side effects in the study.

Table 3 shows the detailed information of each selected industrial app and its applicability on the selected test generation tools. '#Install' shows the number of installs of the app according to Google Play. 'Login' denotes whether logging in is required by the app for its majority functionalities to be available. '#Method' indicates the number of methods in each app as reported by the instrumentation tool, for which '-' indicates that we do not instrument the app for method coverage (more details are available in Section 4.2). '#Activity' shows the number of activities in each app as extracted from `AndroidManifest.xml`. In the 'Availability' header section, 'M.', 'W.', 'Sa.', 'St.', 'D.', and 'A.' stand for Monkey, WCTester, Sapienz, Stoat, DroidBot, and A^{3}E-Depth-First, respectively. Note that the same abbreviation convention is used in subsequent analysis. As shown in the table, most of the selected apps have more than 100 million installs, while each app has at least 1 million installs. These apps span across 30 different categories and are popularly used by Android users everyday. Such factors distinguish these industrial apps from open-source apps, which often have only a few users and very limited functionalities.

4.2 Coverage/Crash Measurement

For code-coverage measurement, we use Ella [2] to instrument all the industrial apps, and collect statistics of method coverage during testing. To avoid potential issues by dual-instrumentation (*i.e.*, instrumentation duplicately conducted by both Ella and a test generation tool under study to collect method coverage), we share the

Ella-collected method coverage information with test generation tools that need the coverage information during testing instead of letting the tools instrument the app again. Note that we focus on coverage of only Java code without considering the native code because Android apps' main functionalities are typically implemented in Java5. In practice, we find that Ella fails to instrument some large industrial apps under study due to the *64K reference limitation* of DalvikVM (see Section 2.2 for details), and some successfully instrumented apps fail to run properly on Android devices due to self-protection mechanisms. To avoid potential bias on app selection caused by instrumentation, we still keep all these apps in the study without collecting their method coverage information. Table 3 also shows whether each app is actually instrumented in the experiments as indicated by the '#Method' column. In addition, we measure activity coverage by periodically monitoring the activity stack on the testing device and extracting all activity names from `AndroidManifest.xml` in each app.

For crash measurement, we monitor the Logcat [19] on target devices during testing and record error messages related to stack traces. We filter out stack traces that are not related to the app under test by checking whether the app's package name is present. Only unique stack traces are counted, achieved by hashing all code locations in each stack trace (instead of the entire stack trace, which might contain environment related information).

4.3 Study Setup

We run each test generation tool continuously for 3 hours on each of their applicable industrial apps under study. Note that for Stoat, we follow the settings described in the tool's corresponding paper [29] by allocating 1 hour for model construction and 2 hours for model evolution. Each test (*i.e.*, a combination of one test generation tool and one applicable app) is run 3 times to compensate potential influence brought by randomness during testing. All tests of the same app are run on the same device. For the fairness of comparison, when we run each test, the tool is allowed to use only one device. For apps requiring logging in to expose most of their functionalities, we choose to manually log in to these apps before each test begins in order to facilitate in-depth testing (note that the code coverage before the test begins is not included in the analysis). In addition, the original implementation of Sapienz clears app data before evaluating each input sequence, reverting the efforts of manual logging in. In order to set up a normalized testing environment while keeping the tool's original design as much as possible, we modify the tool so that it backs up the app data right after manual logging in and later restores the app data instead of clearing them.

We conduct our study on official Android x86 emulators and 4 real phones, all running Android 4.4. Each emulator is configured with 4 CPU cores, 2 GiB of RAM, and 1 GiB of SD card. For each app under study, if the app supports x86 devices, it is tested on a standard emulator each time; otherwise, it is tested on a certain real phone. Apps' data and modifications to the SD card are all reverted after each test. Such design serves as an effort to keep the testing environment efficient, unified, yet versatile to allow testing various industrial apps. Note that Android ARM emulators are not used due to their poor performance, which could potentially limit the power of test generation tools given a bounded amount of time. According to our observation during testing, most x86 emulators seldom use up all dedicated CPU cores, indicating their good performance. Also note that we modify each tool's implementation in only two situations: adapting the tool to our testing environment, or dealing with an easy-to-fix implementation defect that prevents the tool from functioning properly (with reference to the tool's corresponding document or paper).

5 CODE COVERAGE RESULTS ON INDUSTRIAL APPS

In this section, we answer RQ1 (*what is the code coverage achieved by each test generation tool under study on applicable industrial apps*) by measuring and comparing the method and activity coverage achieved by each testing tool on industrial apps in our experiments.

Table 4 shows the statistics of method and activity coverage on each app achieved by each test generation tool under study after 3 hours of testing. '-' in a table cell indicates that the corresponding tool is not applicable on the corresponding industrial app (due to instrumentation or tool applicability issues). Table cells with gray backgrounds indicate the highest values compared with other tools for the same app and coverage type, and multiple tools might have the same highest coverage on the same app (as shown by multiple table cells with gray backgrounds for the same app and coverage type). All coverage percentage numbers are the averaged values of 3 repetitions and are rounded to the nearest integer. Note that due to the number rounding, there might be two tools achieving the same percentage number but only one having the gray background. Also note that we use the same convention in subsequent analysis.

As can be seen from Table 4, Monkey manages to gain the highest method coverage on 22 of 41 apps whose method coverage data can be obtained, although the tool does not achieve much higher method coverage compared with other tools (especially Sapienz) on multiple apps. Sapienz comes after Monkey by gaining the highest method coverage on 14 apps, while other tools perform the best with regard to method coverage on fewer than three apps. Such finding is different from the evaluation results on open-source apps conducted by the authors of some of these tools. According to these authors' evaluation results, they find that their tools achieve higher code coverage on more apps compared with Monkey. It can also be seen that no tool manages to cover more than 50% of methods on any app, with the only exception being Sapienz on the app 'Floor Plan Creator'. In addition, the majority of the tools achieve less than 30% of method coverage on most apps even after 3 hours of testing. Such findings suggest that there is still much space for improving these tools on industrial apps. Another interesting finding is that an app's larger code base is not necessarily more difficult to be covered. For example, the app 'Spotify' has over 200,000 methods, and Sapienz manages to cover 1/3 of these methods. However, the app 'Google Play Music' (abbreviated as 'G.P. Music') has about 23,000 methods, but none of 6 tools cover more than 5% of these methods. Such result also suggests that different industrial apps might have very different characteristics even under the same category.

The statistics of activity coverage are similar to those of method coverage. Monkey gains the highest activity coverage on 35 of all 68 apps (including 3 ties, *i.e.*, there are 3 apps on which Monkey has the same activity coverage as another tool), while Sapienz

Table 4: Statistics of code coverage/fault detection on industrial apps by test generation tools under study

App Name	Method Coverage (%)						Activity Coverage (%)						# of Unique Crashes					
	M.	W.	Sa.	St.	D.	A.	M.	W.	Sa.	St.	D.	A.	M.	W.	Sa.	St.	D.	A.
Abs	26	23	25	15	14	-	13	16	10	0	6	-	5	1	3	0	0	-
AccuWeather	24	18	22	13	18	17	30	14	19	9	16	9	13	1	5	1	2	1
Adobe Acrobat	-	-	-	-	-	-	31	2	36	2	2	5	0	0	3	0	0	0
Amazon Kindle	-	-	-	-	-	-	2	3	1	2	2	1	0	0	0	0	0	0
AutoScout24	22	17	17	19	10	6	8	8	5	13	3	5	2	1	0	14	0	2
Autolist	-	-	-	-	-	-	33	67	50	3	3	3	0	0	1	0	0	1
Best Hairstyles	40	35	40	39	35	9	100	100	100	100	100	25	1	0	0	0	0	0
CNN	32	31	23	22	21	12	48	32	26	26	35	6	4	1	3	0	0	0
Crackle	33	25	32	27	27	27	38	25	38	25	25	19	12	0	11	0	1	0
Duolingo	26	24	26	26	28	25	16	16	15	13	22	9	0	1	2	3	0	0
ES File Explorer	20	21	22	13	10	11	22	20	19	10	6	3	1	1	1	5	0	0
Evernote	23	30	26	15	23	14	11	22	14	3	11	2	0	0	1	0	0	1
Excel	23	16	16	6	15	-	7	4	4	4	4	-	0	1	1	0	0	-
Facebook	-	-	-	-	-	-	4	7	3	1	-	-	3	7	8	1	-	-
Filters For Selfie	50	4	32	1	45	1	50	25	38	13	63	13	9	1	2	0	1	0
Flipboard	32	32	37	28	29	-	12	14	16	8	11	-	4	2	0	4	0	-
Floor Plan Creator	43	36	53	11	32	16	54	38	54	15	31	8	0	0	0	2	0	0
Fox News	-	-	-	-	-	-	29	32	32	3	13	3	5	7	8	0	4	0
G.P. Books	-	-	-	-	-	-	24	15	24	18	15	0	8	2	2	10	1	1
G.P. Music	4	4	4	5	4	3	6	5	3	3	3	2	2	2	4	7	1	1
G.P. Newsstand	5	4	4	4	4	3	6	0	6	0	0	0	1	1	1	1	1	2
GO Launcher Z	23	6	18	11	9	10	14	1	5	1	1	1	0	0	0	0	0	0
Gmail	-	-	-	-	-	-	13	12	17	10	18	-	2	0	3	15	0	-
GoodRx	32	31	31	26	29	22	43	41	31	20	30	7	1	0	17	8	0	1
Google	-	-	-	-	-	-	9	3	4	1	9	-	0	5	1	0	0	-
Google Calendar	-	-	-	-	-	-	22	16	13	9	9	-	7	0	4	14	0	-
Google Chrome	-	-	-	-	-	-	4	2	4	2	2	-	0	0	2	2	0	-
Instagram	-	-	-	-	-	-	25	25	28	10	30	-	3	5	18	0	0	-
LINE Camera	19	28	36	20	7	-	16	28	39	9	9	-	0	0	2	0	0	-
Marvel Comics	19	16	19	14	9	13	50	41	50	30	9	11	5	1	2	9	0	0
Match	10	10	14	12	12	8	9	8	9	3	8	2	0	0	0	0	0	0
McDonald	-	-	-	-	-	-	13	3	15	3	15	8	1	0	0	0	0	0
Merriam-Webster	31	20	34	27	10	19	29	24	24	24	6	6	4	1	4	5	0	0
Messenger	-	-	-	-	-	-	5	9	3	1	1	-	0	0	0	0	0	-
Mirror	22	22	23	21	22	20	33	25	33	17	25	8	4	3	9	5	3	0
My baby Piano	12	3	42	31	30	29	33	33	33	33	33	33	0	0	0	1	0	0
NFL	-	-	-	-	-	-	17	4	13	4	7	4	1	0	1	2	0	1
NOOK	7	3	7	6	13	1	6	2	7	6	12	1	0	0	0	0	0	0
Nike Run Club	-	-	-	-	-	-	30	27	37	1	1	-	3	0	13	0	0	-
OfficeSuite	-	-	-	-	-	-	28	18	11	6	9	1	1	0	0	0	0	0
OneNote	-	-	-	-	-	-	17	20	16	1	13	-	2	0	1	0	0	-
Photos	-	-	-	-	-	-	25	32	25	11	17	-	20	20	13	21	5	-
Pinterest	27	23	26	12	0	6	15	12	21	6	0	3	3	2	3	1	0	0
Quizlet	47	37	46	35	15	32	38	38	40	14	3	9	1	0	1	3	0	0
Sing!	-	-	-	-	-	-	13	19	23	6	15	-	0	0	1	4	0	-
Sketch	-	-	-	-	-	-	37	43	26	13	22	2	8	17	1	5	0	0
Speedometer	29	33	32	24	29	24	73	73	45	18	45	18	2	4	1	0	0	0
Spotify	25	31	33	16	19	-	9	11	12	1	3	-	0	0	0	0	0	-
TED	-	-	-	-	-	-	70	30	56	30	22	15	8	2	2	4	0	0
The Weather Chnl.	-	-	-	-	-	-	9	10	11	10	6	1	1	4	2	5	1	2
Ticketmaster	-	-	-	-	-	-	6	2	7	1	2	-	1	2	1	0	3	-
Translate	32	21	32	14	30	19	58	52	52	12	39	15	0	0	0	2	0	0
TripAdvisor	31	31	29	14	14	1	24	28	24	4	10	0	1	2	5	9	0	1
UC Browser	-	-	-	-	-	-	3	2	3	2	2	2	0	0	0	0	0	0
WEBTOON	26	23	21	19	24	-	50	52	31	16	39	-	1	0	2	1	0	-
WatchESPN	32	21	33	29	13	23	44	31	38	31	13	19	2	0	11	6	0	0
Wattpad	27	37	44	4	30	5	17	32	42	1	16	1	1	2	77	0	0	0
Waze	-	-	-	-	-	-	22	2	8	3	13	1	2	0	0	2	0	0
Wish	33	27	32	21	13	-	35	22	28	5	7	-	0	2	2	0	0	-
Word	23	14	16	6	19	-	7	4	4	0	4	-	0	0	0	0	0	-
Yelp	20	11	20	13	14	4	14	7	12	4	7	0	13	2	26	3	6	2
YouTube	-	-	-	-	-	-	10	6	8	13	2	-	13	2	8	12	0	-
Zedge	36	30	35	21	3	3	23	14	26	6	3	0	12	1	5	9	0	2
Zillow	-	-	-	-	-	-	26	12	20	16	9	4	6	1	2	7	0	1
ibisPaint X	15	18	18	11	16	7	28	28	31	19	31	6	2	3	0	0	2	0
inStar	21	14	21	8	13	3	17	9	17	4	9	4	1	0	1	0	0	1
realtor.com	30	29	30	26	24	19	29	15	24	9	9	6	1	2	1	0	0	0
trivago	40	26	38	18	25	12	41	28	41	17	28	3	1	0	0	5	1	0

gains the highest activity coverage on 28 apps (also including 2 ties). WCTester comes after Sapienz by having the highest activity coverage on 15 apps (including 3 ties). Such finding suggests that WCTester might be better at breadth-first exploration than at in-depth exercising. It can also be seen that, although the overall activity coverage is higher than the method coverage on industrial apps under study, many of the apps still have very low activity coverage. A possible explanation is that many of the apps' main functionalities are actually not reached. Thus, it might be helpful to prioritize unexplored functionalities in order to better saturate the coverage of industrial apps.

To better understand the tools' coverage performance, we investigate into each tool's behavior over time during testing. Figures 1 and 2 show the trend of average method and activity coverage by each test generation tool under study with regard to the elapsed time during testing. Note that we average the coverage percentage numbers of different apps instead of counts of methods or activities to avoid imbalanced influence by apps in different sizes. As shown in Figure 1, Sapienz almost always has the highest average method coverage, although its advantage over Monkey becomes smaller as time goes by. When it comes to the activities, as shown in Figure 2, Monkey constantly has higher average activity coverage than Sapienz. These two tools both have much higher coverage than the remaining four tools. The two tools also gain new coverage faster than all other four tools on average, leading to more significant advantages over time. It can also be seen that A^{3}E-Depth-First (abbreviated as 'A3E') has comparable or higher average coverage with WCTester, Stoat, and DroidBot at the beginning of testing. However, A^{3}E-Depth-First almost stops gaining new coverage after that. According to our observation during testing, such result might be caused by the tool's outdated implementation, which often causes the tool to hang completely (see Section 8 for more discussion).

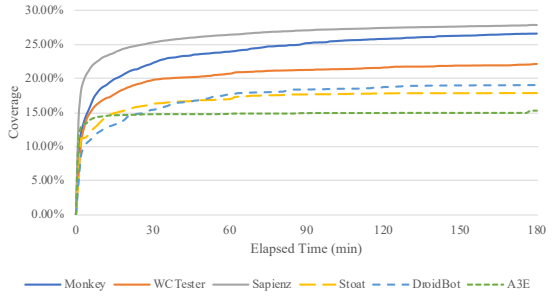

Figure 1: Trend of average method coverage of industrial apps achieved by test generation tools under study

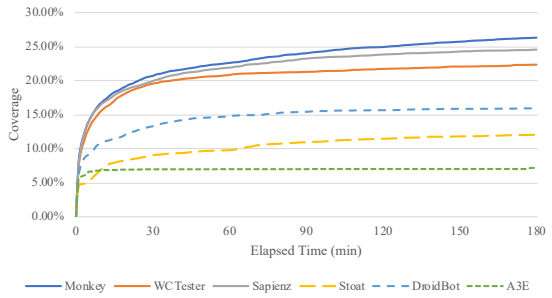

Figure 2: Trend of average activity coverage of industrial apps achieved by test generation tools under study

6 FAULT DETECTION RESULTS ON INDUSTRIAL APPS

In this section, we answer RQ2 (*how many unique crashes can each test generation tool trigger on each applicable industrial app, and what are the causes of these crashes*) by showing the statistics of unique crashes triggered by each testing tool on industrial apps in our experiments.

Table 4 shows the number of unique crashes triggered by each test generation tool on each applicable industrial app under study. Note that each number reports the total number of unique crashes triggered by the tool on the app after 3 repetitions. As shown in Table 4, Stoat triggers the highest numbers of unique crashes on 23 apps, outperforming all other tools. Sapienz triggers the highest numbers of unique crashes on 19 apps, while Monkey accomplishes so on 16 apps. Other three tools trigger the highest numbers of unique crashes on fewer than 10 apps. Also, the numbers of unique triggered crashes have much higher deviations across different tools for the same app, compared with method and activity coverage.

It is somewhat surprising to see that the fault-detection statistics differ from the method and activity coverage statistics. Aiming to understand the differences, we manually investigate into a case involving Stoat and a case involving Sapienz, and examine the details of crashes with the findings as below.

Stoat on the app 'Photos'. Stoat has the highest number of unique crashes on this app. Stoat triggers many `NullPointerExceptions` during starting of activities that take an `Intent` (see Section 2.1 for details) as input. Meanwhile, Monkey and other tools trigger other types of exceptions including `ArrayIndexOutOfBoundsException` and `StackOverflowError`. Stoat's triggering these crashes during activity starting might benefit from injecting system-level events during testing.

Sapienz on the app 'Wattpad'. This combination has much more unique crashes than any other combinations. We find that Sapienz triggers numerous `SQLiteExceptions` on this app for each of the three runs. The exception causes are mostly about querying on multiple non-existent tables in the app's SQLite database. As the app seems to heavily rely on the SQLite database but does not properly handle related exceptions, these fatal SQL queries are frequently triggered from multiple locations of the app, causing different stack traces. None of other tools is able to trigger such number of exceptions during testing. A possible explanation is that triggering such crashes requires special preconditions (*e.g.*, forcibly terminating the app during initialization, which involves SQL operations for creating these tables) that other tools might not be able to create.

7 RANK-1 ANALYSIS ON EXPERIMENT RESULTS

In this section, in order to provide additional insights for answering RQ3 (*how to efficiently combine multiple test generation tools on applicable industrial apps to achieve better coverage and fault detection*), we measure and analyze the statistics of rank-1 method and activity coverage plus rank-1 unique crashes achieved by each test generation tool on industrial apps in our experiments. We also analyze the results from previous sections to answer RQ3.

A rank-n method/activity or unique crash indicates that there are n test generation tools being able to cover the method/activity or unique crash [21]. Specifically, a rank-1 method/activity or unique crash indicates that only one test generation tool under study covers the method/activity or trigger the unique crash in at least one run of our experiments. For each tool under study, the numbers of its covered rank-1 methods/activities and triggered rank-1 unique crashes reflect the tool's unique value to testing an app.

Table 5 shows the statistics of rank-1 methods, activities, and unique crashes on applicable industrial apps by the test generation tools under study. A table cell with 'm/n' indicates that, on the corresponding app, the *rank-1* methods/activities or unique crashes covered by the corresponding test generation tool account for m percent of covered methods/activities or triggered unique crashes by all the six test generation tools, and *all* of the tool's covered methods/activities or triggered unique crashes are n percent of covered methods/activities or triggered unique crashes by all the six test generation tools. With such definition, we know that on a specific app, if test generation tool A's method/activity or unique crash statistic is 'a/b' and tool B's method/activity or unique crash statistic is 'c/d', by running both tool A and tool B (*i.e.*, combining tool A and tool B) we could achieve at least $\max(a+d, b+c)$ percent coverage of methods/activities or unique crashes that are covered or triggered by all the six test generation tools.

As shown in Table 5, for many industrial apps under study, combining Monkey and Sapienz facilitates good saturation of covering the app code as they together contribute to over 90% of all covered methods by all the six tools on these apps. These two tools also have the highest numbers of rank-1 covered methods on many apps. When it comes to activities, combining Monkey with Sapienz

Table 5: Statistics of rank-1 methods, activities, and unique crashes on industrial apps by test generation tools under study

App Name	% of Rank-1 Covered Methods						% of Rank-1 Covered Activities						% of Rank-1 Unique Crashes					
	M.	W.	Sa.	St.	D.	A.	M.	W.	Sa.	St.	D.	A.	M.	W.	Sa.	St.	D.	A.
Abs	4/76	16/88	1/74	0/46	0/63	-	0/57	43/100	0/57	0/14	0/29	-	56/56	11/11	33/33	0/0	0/0	-
AccuWeather	5/98	0/75	1/93	1/51	0/74	0/66	33/83	0/39	0/50	17/44	0/44	0/22	47/87	0/7	0/33	7/7	0/13	7/7
Adobe Acrobat	-	-	-	-	-	-	6/88	0/12	12/94	0/12	0/12	0/12	0/0	0/0	100/100	0/0	0/0	0/0
Amazon Kindle	-	-	-	-	-	-	14/50	43/79	0/7	0/21	0/14	0/7	0/0	0/0	0/0	0/0	0/0	0/0
AutoScout24	9/91	1/66	0/73	3/81	3/46	0/23	9/36	0/36	0/36	55/82	0/9	0/18	11/11	5/5	0/0	74/74	0/0	11/11
Autolist	-	-	-	-	-	-	0/73	9/91	0/82	0/5	0/5	0/5	0/0	0/0	50/50	0/0	0/0	50/50
Best Hairstyles	0/95	0/95	0/96	0/95	2/87	0/30	0/100	0/100	0/100	0/100	0/100	0/50	100/100	0/0	0/0	0/0	0/0	0/0
CNN	6/91	1/91	1/62	0/65	0/81	0/34	13/100	0/75	0/69	0/63	0/81	0/13	50/50	13/13	38/38	0/0	0/0	0/0
Crackle	1/98	0/78	1/97	0/86	0/92	0/82	0/86	0/57	0/86	14/71	0/57	0/57	43/57	0/0	43/52	0/0	0/5	0/0
Duolingo	0/90	1/93	1/88	2/90	2/93	0/87	6/65	6/71	0/59	12/65	0/71	0/47	0/0	17/17	33/33	50/50	0/0	0/0
ES File Explorer	4/79	6/82	4/76	4/55	0/54	0/37	5/74	0/69	3/64	13/51	0/33	0/8	13/13	13/13	13/13	63/63	0/0	0/0
Evernote	3/75	7/86	4/78	0/43	1/71	0/35	2/55	14/86	6/61	0/18	0/55	0/6	0/0	0/0	50/50	0/0	0/0	50/50
Excel	17/98	1/77	1/69	0/25	0/67	-	50/100	0/50	0/50	0/50	0/50	-	0/0	50/50	50/50	0/0	0/0	-
Facebook	-	-	-	-	-	-	9/38	50/83	0/36	0/9	-	-	11/17	33/39	44/44	6/6	-	-
Filters For Selfie	8/96	0/8	0/84	0/2	4/86	0/2	0/80	0/40	0/80	0/20	20/100	0/20	80/90	0/10	10/20	0/0	0/10	0/0
Flipboard	2/72	2/76	11/88	1/65	1/66	-	4/52	0/61	22/70	4/39	0/39	-	40/40	20/20	0/0	40/40	0/0	-
Floor Plan Creator	2/79	0/63	11/95	1/36	1/64	0/40	0/88	0/75	0/88	13/63	0/63	0/25	0/0	0/0	0/0	100/100	0/0	0/0
Fox News	-	-	-	-	-	-	8/85	8/92	0/85	0/8	0/23	0/8	7/36	36/50	14/57	0/0	0/29	0/0
G.P. Books	-	-	-	-	-	-	0/80	0/60	0/80	20/80	0/60	0/0	26/42	0/11	0/11	53/53	0/5	5/5
G.P. Music	9/86	0/89	1/90	0/87	0/88	0/74	0/100	0/100	0/50	0/50	0/50	0/25	8/17	8/17	25/33	50/58	0/8	0/8
G.P. Newsstand	19/96	0/78	1/78	0/78	0/74	0/74	0/100	0/0	0/100	0/0	0/0	0/0	0/50	0/50	0/50	0/50	0/50	50/100
GO Launcher Z	31/89	0/29	5/63	0/36	4/48	0/35	62/88	0/3	6/32	0/3	6/9	0/3	0/0	0/0	0/0	0/0	0/0	0/0
Gmail	-	-	-	-	-	-	0/60	0/47	0/73	20/47	7/80	-	5/11	0/0	11/16	79/79	0/0	-
GoodRx	1/96	0/94	1/93	0/86	0/89	0/66	0/81	3/76	0/51	14/62	0/62	0/11	0/4	0/0	62/65	31/31	0/0	4/4
Google	-	-	-	-	-	-	13/87	0/67	0/47	0/7	0/73	-	0/0	83/83	17/17	0/0	0/0	-
Google Calendar	-	-	-	-	-	-	0/78	0/56	0/56	0/33	11/67	-	28/28	0/0	16/16	56/56	0/0	-
Google Chrome	-	-	-	-	-	-	0/75	0/50	0/75	25/75	0/50	-	0/0	0/0	50/50	50/50	0/0	-
Instagram	-	-	-	-	-	-	0/81	0/81	0/81	13/38	0/75	-	4/13	17/22	65/78	0/0	0/0	-
LINE Camera	1/66	5/81	15/92	0/53	0/29	-	4/61	0/75	21/96	0/21	0/21	-	0/0	0/0	100/100	0/0	0/0	-
Marvel Comics	2/91	0/80	5/94	3/78	0/45	0/61	3/79	0/66	0/76	17/76	0/21	0/17	20/33	0/7	7/13	60/60	0/0	0/0
Match	0/95	0/84	2/97	0/83	0/85	2/54	0/100	0/83	0/100	0/33	0/83	0/17	0/0	0/0	0/0	0/0	0/0	0/0
McDonald	-	-	-	-	-	-	14/86	0/14	0/64	0/21	0/64	14/57	100/100	0/0	0/0	0/0	0/0	0/0
Merriam-Webster	1/83	0/83	15/91	0/68	0/26	0/51	0/75	0/75	13/63	13/50	0/13	0/13	29/29	7/7	29/29	36/36	0/0	0/0
Messenger	-	-	-	-	-	-	0/37	53/95	5/26	0/11	0/11	-	0/0	0/0	0/0	0/0	0/0	-
Mirror	0/94	1/94	3/98	1/92	0/92	0/83	0/100	0/75	0/100	0/75	0/75	0/25	0/31	0/23	31/69	23/38	0/23	0/0
My baby Piano	10/25	0/6	19/87	2/67	0/63	0/61	0/100	0/100	0/100	0/100	0/100	0/100	0/0	0/0	0/0	100/100	0/0	0/0
NFL	-	-	-	-	-	-	45/100	0/36	0/55	0/36	0/45	0/18	20/20	0/0	20/20	40/40	0/0	20/20
NOOK	0/40	0/21	2/41	5/48	46/79	0/5	0/32	0/20	8/40	16/56	32/76	0/4	0/0	0/0	0/0	0/0	0/0	0/0
Nike Run Club	-	-	-	-	-	-	4/84	0/71	12/88	0/2	0/6	-	13/20	0/0	80/87	0/0	0/0	-
OfficeSuite	-	-	-	-	-	-	22/84	8/67	4/31	0/24	0/24	0/2	100/100	0/0	0/0	0/0	0/0	0/0
OneNote	-	-	-	-	-	-	6/83	6/83	6/72	0/6	0/56	-	67/67	0/0	33/33	0/0	0/0	-
Photos	-	-	-	-	-	-	2/77	2/89	0/75	7/55	0/55	-	15/30	18/30	17/20	30/32	3/8	-
Pinterest	6/81	12/85	2/77	0/38	0/0	0/19	0/55	9/64	9/64	9/36	0/0	0/9	33/33	22/22	33/33	11/11	0/0	0/0
Quizlet	1/88	1/69	7/95	1/67	0/28	0/70	0/64	0/69	25/92	3/28	0/6	0/25	20/20	0/0	20/20	60/60	0/0	0/0
Sing!	-	-	-	-	-	-	0/44	11/78	6/83	6/28	0/56	-	0/0	0/0	20/20	80/80	0/0	-
Sketch	-	-	-	-	-	-	0/64	18/82	0/50	7/32	4/46	0/4	26/26	55/55	3/3	16/16	0/0	0/0
Speedometer	2/87	0/90	7/87	0/63	0/79	0/72	0/100	0/100	0/75	0/25	0/63	0/38	0/50	25/100	0/25	0/0	0/0	0/0
Spotify	6/91	1/87	3/91	0/44	0/66	-	22/78	4/65	4/65	0/4	0/17	-	0/0	0/0	0/0	0/0	0/0	-
TED	-	-	-	-	-	-	4/83	0/50	0/67	17/67	0/21	0/21	47/53	13/13	7/13	27/27	0/0	0/0
The Weather Chnl.	-	-	-	-	-	-	11/59	7/52	0/48	26/63	0/26	0/4	0/8	15/31	15/15	38/38	0/8	15/15
Ticketmaster	-	-	-	-	-	-	0/20	0/20	80/100	0/10	0/20	-	0/33	0/67	0/33	0/0	0/100	-
Translate	2/96	0/89	1/96	0/57	1/90	0/56	0/95	0/90	0/100	0/15	0/75	0/25	0/0	0/0	0/0	100/100	0/0	0/0
TripAdvisor	7/88	4/85	2/81	0/43	0/38	0/0	6/78	8/86	3/74	1/19	0/34	0/0	6/6	6/12	24/29	53/53	0/0	6/6
UC Browser	-	-	-	-	-	-	20/60	0/40	40/80	0/20	0/20	0/20	0/0	0/0	0/0	0/0	0/0	0/0
WEBTOON	12/96	0/68	2/76	0/63	0/72	-	3/97	0/92	0/71	0/39	0/63	-	25/25	0/0	50/50	25/25	0/0	-
WatchESPN	1/96	0/63	3/98	1/94	0/37	0/68	11/89	0/56	0/78	11/78	0/22	0/33	11/11	0/0	58/58	32/32	0/0	0/0
Wattpad	4/76	1/83	7/92	0/8	1/65	0/14	5/39	4/65	16/86	0/2	5/35	0/2	1/1	3/3	96/96	0/0	0/0	0/0
Waze	-	-	-	-	-	-	45/93	0/9	2/32	0/20	2/52	0/9	50/50	0/0	0/0	50/50	0/0	0/0
Wish	12/90	2/77	4/85	0/56	0/38	-	25/86	3/53	8/72	0/17	0/19	-	0/0	50/50	50/50	0/0	0/0	-
Word	6/94	3/88	1/68	0/26	1/81	-	0/100	0/100	0/50	0/50	0/50	-	0/0	0/0	0/0	0/0	0/0	-
Yelp	16/94	1/36	2/77	1/61	1/66	0/17	31/87	4/26	1/54	1/24	1/36	0/3	12/32	5/5	39/63	0/7	12/15	5/5
YouTube	-	-	-	-	-	-	18/55	0/27	0/36	45/64	0/9	-	29/42	6/6	13/26	39/39	0/0	-
Zedge	6/96	0/81	2/90	1/58	0/8	0/11	8/83	0/50	8/83	8/42	0/8	0/8	26/52	0/4	0/22	39/39	0/0	9/9
Zillow	-	-	-	-	-	-	20/77	9/46	0/51	9/43	3/20	0/11	33/40	0/7	7/13	40/47	0/0	7/7
ibisPaint X	4/86	1/81	3/73	1/67	0/79	0/29	6/94	0/81	0/69	0/63	0/81	0/13	20/40	40/60	0/0	0/0	0/40	0/0
inStar	1/96	0/69	3/98	0/55	0/60	0/16	0/80	0/40	20/100	0/40	0/40	0/20	33/33	0/0	33/33	0/0	0/0	33/33
realtor.com	5/94	1/79	2/90	0/76	0/66	0/52	36/100	0/36	0/64	0/29	0/21	0/14	33/33	33/67	0/33	0/0	0/0	0/0
trivago	4/96	0/60	1/93	2/58	0/65	0/29	0/65	5/50	0/65	30/60	0/45	0/5	14/14	0/0	0/0	71/71	14/14	0/0

and/or Stoat seems to be a good option for most of the apps, because Monkey has the highest numbers of covered activities (regardless of ranking) on many apps while Sapienz and/or Stoat can be good complements when Monkey is not able to cover most activities. For fault detection, combining Stoat with Sapienz and/or Monkey seems to be more effective for most of the apps, as Stoat has the highest numbers of unique crashes (regardless of ranking) on many apps while Sapienz and/or Monkey can be good complements. Such suggestion is consistent with the results of manual investigation from Section 6, where we find that Stoat and Sapienz/Monkey can trigger very different types of crashes. Also, according to the fact that WCTester is designed for WeChat, the tool might be a good complement with Monkey when the app under test involves similar scenarios as those in WeChat (*e.g.*, chatting, social, and information browsing). Rank-1 activity statistics also show hints on this suggestion: WCTester covers the highest numbers of unique activities on 'Facebook', 'Messenger', 'Pinterest', and 'TripAdvisor'. All these apps share similar usage scenarios with some functionalities of WeChat.

8 EXPERIENCE IN APPLYING TEST GENERATION TOOLS ON INDUSTRIAL APPS

In this section, we answer RQ4 (*how much effort does it require to set up each test generation tool for testing industrial apps*) by reporting our experience on setting up each test generation tool under study and applying them on selected industrial apps. We additionally report our experience with Ella [2] and the Android Framework (illustrated in Section 2).

8.1 Test Generation Tools

Monkey. As the built-in test generation tool shipped with each Android device, Monkey can be invoked directly using the Android Debug Bridge [16] shell interface. We spend no effort on setting up Monkey for industrial apps under study.

WCTester. Due to defects in the UIAutomator Python wrapper [20] being used, WCTester often halts during exploration and produces error messages such as "RPC server not connected". We spend about 5 hours investigating and fixing the defects, and after that WCTester becomes much more stable.

Sapienz. The original implementation of Sapienz supports only emulators. Given that many apps under study include native libraries compiled against only ARM processors, we modify the tool's implementation to add support for real devices. Since the tool is tested on only Android 4.4 and needs to install `MotifCore` to the system partition, for maximum compatibility, we downgrade all real devices to Android 4.4 and acquire the root privileges on all of them. We also modify the tool's implementation so that it restores the app data to the point right after manual logging in instead of clearing them. Finally, we spend more than 10 hours getting Sapienz to work in our settings.

Stoat. The original implementation of Stoat has multiple issues with our testing infrastructure. For example, it forcibly kills all Java and ADB processes on the underlying computer to clean up the environment, unexpectedly terminating our tools for monitoring the testing. Stoat also uses the problematic UIAutomator Python wrapper. Overall, we spend about 10 hours investigating and fixing the implementation of Stoat.

DroidBot. DroidBot needs to run its client app under the accessibility mode, which requires granting the privilege manually in Android system settings. We also sometimes encounter error messages such as "Please enable DroidBot manually in accessibility settings" even if the tool works in previous runs. Overall, we spend about 2 hours writing a script to mitigate this issue.

A^3E-Depth-First. A^3E-Depth-First has several issues in the implementation, such as not being able to click buttons with labels containing special characters. Due to the outdated implementation and the need of running the target app under its instrumentation, the tool causes many apps to crash at beginning, preventing them from being tested. It also hangs during exploration for unknown reasons even after we try to fix this issue. We spend about 5 hours trying to fix the issues for the tool.

Note that we have already submitted bug reports on most of the preceding patches to these existing tools for the original tool authors to improve the quality and robustness of these tools. Additionally, due to the fact that some tool issues appear only after the experiments have lasted for some time, it takes a lot of manual efforts to inspect the experiment results to find out such issues, and the wasted time of running these experiments (with these issues still existing in the tools) adds up to tens of hours.

8.2 Ella and the Android framework

Ella. Ella has multiple implementation issues in different modules. In addition, the tool's original implementation does not support instrumenting apps with *multidex* enabled, which is commonly used by large industrial apps. We spend more than 10 hours fixing the issues and adding *multidex* support to Ella.

Android framework. We even encounter an issue in the system framework on Android 4.4. Specifically, the issue in the UIAutomator framework causes the service to stop working when there is any special character (*e.g.*, an Emoji icon) on the screen. We fix the issue by modifying the corresponding Android source code plus recompiling and replacing the UIAutomator framework (`uiautomator.jar`). We spend about 5 hours addressing this issue.

9 THREATS OF VALIDITY

The main threat to external validity is the representativeness of the studied subjects (*i.e.*, the degree to which the studied industrial apps and tools are representative of true practice). Our current tool set contains only six test generation tools due to not being able to apply other test generation tools on most industrial apps under study. However, these six tools are state-of-the-art ones that are already compared with more state-of-the-art or state-of-the-practice tools such as Monkey, which is popularly used in industry. These threats could be reduced by more experiments on wider types of subjects in future work.

The threats to internal validity are instrumentation effects that can bias our results. Issues in Ella's handling of the apps' binary code, faults in our modification of the existing tools or in our experiment scripts, etc. might cause such effects. To reduce these threats, we manually inspect traces of our experiments for sample apps. In addition, we are not able to obtain method coverage for about half of the industrial apps under study due to Ella's failing to instrument these apps or these apps not running normally after instrumentation. We also try coverage collection tools based on Soot [30] and they simply fail or cause problems on more apps. We are not aware of other tools that can flawlessly instrument these large, complex, and closed-source apps. Also, it might cause bias to the selection of apps if we simply discard these apps that fail to be instrumented.

10 CONCLUSION

In this paper, we have presented an empirical study of existing Android test generation tools' applicability on industrial apps. We directly compare the tools with regard to code coverage and fault-detection ability. By analyzing the study results, we provide suggestions for combining different test generation tools to achieve better performance. We also report our experience in applying these tools to industrial apps under study. Our study results give insights on how Android UI test generation tools could be improved to better handle industrial apps.

Our study results offer a strong implication that testing researchers for Android test generation tools should empirically compare a newly proposed tool with related previous tools on industrial apps ***besides*** open-source apps, going *beyond* the current common research practice of comparing tools on *only* open-source apps.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation under grants no. CNS-1513939 and CNS-1564274.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 258–261. https://doi.org/10.1145/2351676.2351717

[2] Saswat Anand. 2016. ELLA: A Tool for Binary Instrumentation of Android Apps. https://github.com/saswatanand/ella

[3] 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)*. ACM, New York, NY, USA, Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666

[4] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. 2003. An Introduction to MCMC for Machine Learning. *Machine Learning* 50, 1 (01 Jan 2003), 5–43. https://doi.org/10.1023/A:1020281327116

[5] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87 – 106. https://doi.org/10.1016/0890-5401(87)90052-6

[6] 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 '13)*. ACM, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[7] 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)*. ACM, New York, NY, USA, 623–640. https://doi.org/10.1145/2509136.2509552

[8] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 429–440. https://doi.org/10.1109/ASE.2015.89

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp* 6, 2 (April 2002), 182–197. https://doi.org/10.1109/4235.996017

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[11] Google. 2017. Android Dalvik Executable format. https://source.android.com/devices/tech/dalvik/dex-format

[12] Google. 2017. ART and Dalvik. https://source.android.com/devices/tech/dalvik/

[13] Google. 2018. Android 64K Method limit. https://developer.android.com/studio/build/multidex

[14] Google. 2018. Android App Components. https://developer.android.com/guide/components/fundamentals#Components

[15] Google. 2018. Android Apps on Play Store. https://play.google.com/store/apps

[16] Google. 2018. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb

[17] Google. 2018. Android Monkey. https://developer.android.com/studio/test/monkey

[18] Google. 2018. Android Platform Architecture. https://developer.android.com/guide/platform/

[19] Google. 2018. Logcat command-line tool. https://developer.android.com/studio/command-line/logcat

[20] Xiaocong He. 2018. Python wrapper of Android uiautomator test tool. https://github.com/xiaocong/uiautomator

[21] K. Inkumsah and T. Xie. 2008. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 297–306. https://doi.org/10.1109/ASE.2008.40

[22] Business Insider. 2018. WeChat has hit 1 billion monthly active users. http://www.businessinsider.com/wechat-has-hit-1-billion-monthly-active-users-2018-3

[23] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and Replay for Android: Are We There Yet in Industrial Cases?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. ACM, New York, NY, USA, 854–859. https://doi.org/10.1145/3106237.3117769

[24] 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 Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[25] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2018. DroidBot: A lightweight test input generator for Android. https://github.com/honeynet/droidbot

[26] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[27] 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 '16)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[28] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd Intelligence Enhances Automated Mobile Testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. IEEE Press, Piscataway, NJ, USA, 16–26. http://dl.acm.org/citation.cfm?id=3155562.3155569

[29] 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 '17)*. ACM, New York, NY, USA, 245–256. https://doi.org/10.1145/3106237.3106298

[30] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. (1999), 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[31] 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)*. Springer-Verlag, Berlin, Heidelberg, 250–265. https://doi.org/10.1007/978-3-642-37057-1_19

[32] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, New York, NY, USA, 987–992. https://doi.org/10.1145/2950290.2983958

[33] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 253–262. https://doi.org/10.1109/ICSE-SEIP.2017.32