EMPOWERING AUTOMATED MOBILE UI TESTING WITH EXTERNAL SUPPORT

BY

WENYU WANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Tao Xie, Chair
Professor Darko Marinov
Assistant Professor Tianyin Xu
Dr. Mukul Prasad, Fujitsu Research of America

ABSTRACT

While mobile devices have become an integral part of modern daily life, the ever-growing complexity and fast pace of app feature development have imposed unprecedented challenges on making these mobile apps robust and reliable. User Interface (UI), as the primary medium of user interactions, is naturally a good entry point for testing mobile apps. While manual and scripted UI testing is a common practice, automated UI testing is becoming increasingly popular. By mimicking how human users interact with apps through the UIs, automated UI test generation tools are capable of detecting reliability and usability issues, complementing manual and scripted testing by requiring little to no human testing effort.

There have been numerous mobile UI test generation tools from both the research community and industry after years of development, mainly focusing on designing novel exploration algorithms on the Android platform. Despite showing the best overall test effectiveness in their own evaluation settings, most of the existing tools are found to barely outperform a baseline tool, Monkey, when evaluated upon comprehensive sets of Android apps. The observation is made by independent measurement studies (including one described in this dissertation) involving both relatively simple open-source apps and popular, complex industrial apps. The finding, contradicting researchers' common belief, suggests a significant effectiveness gap that needs to be filled for automated mobile UI testing, especially on industrial apps with generally high impacts.

Aiming to understand the aforementioned effectiveness gap, we empirically investigate the test process and results from our measurement study, yielding three relevant findings: (1) there is no "silver-bullet" tool that outperforms all other tools on every app, suggesting that it is difficult to build a single tool that adapts well to different apps with diverse UI designs; (2) a tool's test effectiveness is not solely decided by its exploration algorithm, and the tool's implementation also makes differences; (3) it is possible to enhance the design or the implementation of different tools using unified approaches. In the context of existing work's focus on designing novel exploration algorithms, our findings suggest that it is worthwhile to develop complementary techniques that enhance existing tools to unleash the power of different exploration algorithms on various complex industrial apps.

Inspired by the aforementioned findings, this dissertation presents three parts of research that explore the possibilities for *existing* automated UI test generation tools to be empowered with *external* automated support (i.e., techniques that are applicable to various tools while keeping them fully automatic). These parts of work enhance different components in the workflow of automated Android UI test generation tools. The first part (Toller) focuses on enhancing *infrastructure support* that enables a tool's exploration algorithm to obtain states from and execute actions on the test device, allowing the tool to iterate faster and cover more App Under Test

(AUT) functionalities within limited time. The second part (VET) focuses on providing *exploration guidance* for a tool on a specific AUT, based on our observation that a tool's exploration algorithm or implementation might have applicability issues in certain conditions. The third part (EPIT) focuses on *parallelization coordination* for a tool and a specific AUT on multiple test devices to improve the overall test effectiveness or reduce testing costs by reducing overlapped explorations. Our evaluations show that the proposed techniques can help state-of-the-art automated Android UI test generation tools achieve substantially better test effectiveness or reduce testing costs on popular and complex industrial apps.

To my family and friends, for their love and support.

ACKNOWLEDGMENTS

Five years of Ph.D. study at UIUC has been a long journey in my life. I am very fortunate to have chances to work with or get help from many bright minds around me. In this acknowledgement, please allow me to express my deepest gratitude for all of the support that I have received over years.

I would first like to thank my advisor, Professor Tao Xie, and my co-advisor, Professor Tianyin Xu. It is a great privilege to work with both of them. Tao has provided me with enormous and continuous guidance throughout my Ph.D. study, and I truly appreciate the opportunity offered by him for me to come to UIUC and work with him. Research-wise, Tao has always been open to my wild ideas while constantly reminding me of critical thinking and challenging me with in-depth questions, encouraging me to become an independent researcher. Life-wise, Tao is always my source of invaluable lessons. Our group motto, *work hard, work smart, work wise*, inspires me to strive, think, and improve all the time. Tianyin has provided me with his great support and supervision for the past three years. He is always willing and passionate to discuss with me on research projects and life experiences. His positivity has helped me through the toughest times of my Ph.D. study. It is always fascinating to hear from him about the systems research that I have also been interested in for quite a while. I truly enjoy and appreciate our friend-like relationship.

I would then like to offer special thanks to the other two members of my thesis committee, Professor Darko Marinov and Dr. Mukul Prasad. Both of them have been very supportive of my dissertation work. Darko has kindly provided me with lots of help and advice throughout my Ph.D. study. Even if we have never collaborated on any project, he is always happy to hear about my research and offer his thoughts and insights. Mukul has also been a great source of valuable feedback on my work. I truly appreciate the opportunity to work with him as a summer intern back in 2018.

I would also like to thank many group/lab mates with whom I am fortunate to spend my last several years. First, I hope to thank the members of the group led by Professor Tao Xie, at both UIUC and PKU. Specifically, I thank Angello Astorga, Liia Butler, Joey (Jiayi) Cao, Tianyu Chen, Yingjie Fu, Shirdon Gorse, Yibo He, Junhao Hu, Ziyue Hua, Lori Jia, Wing Lam, Jade (Yue) Leng, Linyi Li, Zhenwen Li, Zongyang Li, Wei Lin, Chenxu Liu, Xueqing Liu, Jonathan Osei-Owusu, Dezhi Ran, Luyao Ren, Siwakorn Srisakaokul, Zhengkai Wu, Wei Yang, Hao Yu, Mingming Zhang, Bingchan Zhao, Zelin Zhao, Zirui Neil Zhao, and Zexuan Zhong, for their companion during many group meetings and activities. Then, I would like to thank the members of the groups led by Professor Sasa Misailovic and Professor Tianyin Xu. Specifically, I thank Jack Chen, Qingrong Chen, Sam Cheng, Saikat Dutta, Vimuth Fernando, Zixin Huang, Keyur Joshi, Jacob Laurel, Xudong Sun, Andrew B. Yoo, and Yifan Zhao, for discussions about research, courses, and life, as

well as many activities that we had together. Last, I would like to thank the senior students who offered me with their valuable help on my research life, Ph.D. exams, and job finding. Specifically, I thank Alex Gyori, Wajih Ul Hassan, Chiao Hsieh, Owolabi Legunsen, August Shi, and Qi Wang.

Many thanks to my academic collaborators over the years: Benjamin Andow, Yurui Cao, Wontae Choi, Jing Cui, Yuetang Deng, William Enck, Pinjia He, Hui Jin, Wing Lam, Yue Leng, Dengfeng Li, Zongyang Li, Chenxu Liu, Dian Liu, Xueqing Liu, Hui Luo, Weizhi Meng, George Necula, Mukul R. Prasad, Dezhi Ran, Bradley Reaves, Ripon K. Saha, Koushik Sen, Kapil Singh, Zihe Song, Xing Tang, Justin Whitaker, Xionglin Wu, Zhengkai Wu, Tao Xie, Tianyin Xu, Peng Yan, Wei Yang, Samin Yaseer Mahmud, Qinsong Zeng, Chengxiang Zhai, Changrong Zhang, Zhenwen Zhang, Haibing Zheng, and Wujie Zheng. I would not be able to finish all the work in this dissertation without your help. I sincerely hope that there will be chances for us to collaborate again.

I would also like to specifically thank my research mentors from my days as an undergraduate student: Professor Koushik Sen and Wontae Choi from UC Berkeley, with whom I worked on my first-ever Software Engineering research project [1] during my visit to Berkeley back in 2016, as well as Professor Peng Wang from SEU, who led me to the world of CS research. I would never have the motivation or the destiny to start my Ph.D. study without the guidance or the opportunities from them.

I would finally like to thank my family, especially my parents. They are always supportive of my own decisions and choices while being my strongest backing. I cannot say enough about my gratitude for everything that they have done for me.

I apologize to those who have helped me but are not explicitly mentioned here. Nevertheless, please understand that you all have my deepest gratitude for what you have done for me over the years.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Thesis Statement 3
1.2 Contributions 4
1.3 Dissertation Organization 6

CHAPTER 2 UNDERSTANDING THE EFFECTIVENESS OF ANDROID UI TEST GENERATION TOOLS ON INDUSTRIAL APPS 7
2.1 Overview 7
2.2 Background 9
2.3 Selection of Android Test Generation Tools 11
2.4 Study Methodology 13
2.5 Code Coverage Results on Industrial Apps 17
2.6 Fault Detection Results on Industrial Apps 20
2.7 Rank-1 Analysis on Experiment Results 22
2.8 Experience in Applying Test Generation Tools on Industrial Apps 24
2.9 Threats of Validity 26
2.10 Summary 27

CHAPTER 3 TOLLER: ENHANCING INFRASTRUCTURE SUPPORT BY SYSTEM DESIGN 28
3.1 Overview 28
3.2 Background 29
3.3 Motivating Study 32
3.4 Design and Implementation of Toller 35
3.5 Evaluation 37
3.6 Threats to Validity 45
3.7 Discussion 46
3.8 Summary 47

CHAPTER 4 VET: PROVIDING EXPLORATION GUIDANCE VIA IDENTIFYING AND AVOIDING UI EXPLORATION TARPITS 48
4.1 Overview 48
4.2 Motivating Examples 50
4.3 Background 53
4.4 The Vet Approach 54
4.5 Evaluation 60
4.6 Discussion and Limitations 69
4.7 Threats to Validity 70
4.8 Summary 70

CHAPTER 5 EPIT: FACILITATING PARALLELIZATION COORDINATION WITH UI EXPLORATION SPACE PARTITIONING 72
5.1 Overview 72
5.2 Motivating Example 74
5.3 The Epit Approach 75
5.4 Evaluation 81
5.5 Discussion and Limitations 89
5.6 Threats to Validity 90
5.7 Summary 90

CHAPTER 6 RELATED WORK 92

CHAPTER 7 CONCLUSIONS AND FUTURE WORK 96
7.1 Future Work 97

REFERENCES 99

CHAPTER 1: INTRODUCTION

Mobile devices have become an integral part of modern daily life, especially in the pandemic era (e.g., food ordering, grocery delivery, and social networking). The ever-growing complexity and fast pace of app feature development have imposed unprecedented challenges on making these mobile apps robust and reliable [2]. User Interface (UI), as the primary medium of user interactions, is naturally a good entry point for testing mobile apps. UIs generally expose vast functionalities through unified interfaces, making them a good fit for automated testing. While manual and scripted UI testing is a common practice, automated UI testing is becoming increasingly popular [3, 4, 5, 6]. Automated UI test generation tools are capable of detecting reliability and usability issues, complementing manual and scripted testing by requiring little to no human testing effort. Developers and testers can effortlessly run automated UI test generation tools anytime, for long periods of time, and for multiple apps across many devices. Besides aiming to achieve comparable coverage of app functionalities with human efforts, these tools can also help with more thoroughly covering app logic that could be overlooked by human testers.

Automated UI test generation tools work by mimicking how human users interact with apps through the UIs, where Figure 1.1 shows a simplified view of these tools' workflow in general. Specifically, the entire testing process is driven by the *exploration algorithm*, which is the core component of an automated UI test generation tool. The exploration algorithm repeatedly interacts with the *test device*, where the App Under Test (AUT) is installed and running. At each step, the exploration algorithm first obtains information about the current state of the test device and AUT (e.g., screen contents). Based on the currently and historically observed status, the exploration algorithm then decides which UI action (e.g., a screen tap or a key press) or system action (e.g., app restarting) to perform on the test device. A tool usually generates test logs or reports along with the testing process to help developers/testers understand how the tool exercises the AUT. Additionally, developers/testers can measure the test effectiveness of a tool by monitoring the testing process, usually by collecting the code coverage and crash triggering statistics.

There have been numerous mobile UI test generation tools from both the research community and industry after years of development, mainly focusing on designing novel exploration algorithms on the Android platform. *Monkey* [7], an Android UI test generation tool developed by Google, is one of the earliest efforts in this direction. The tool produces purely randomized UI event sequences and efficiently injects them into the target Android system without considering the design details of the AUT. Developed after the release of Monkey, various test generation tools aim to improve Monkey's simple exploration algorithm. They are mainly randomness-driven/evolutionary-algorithm-based tools [8, 9, 10], model-based tools [11, 12, 13, 14, 15], and systematic-exploration-based tools [16, 17, 18]. In each tool's own evaluation setting, the tool authors find that their tool achieves better overall test effectiveness compared with Monkey (and often the time, other

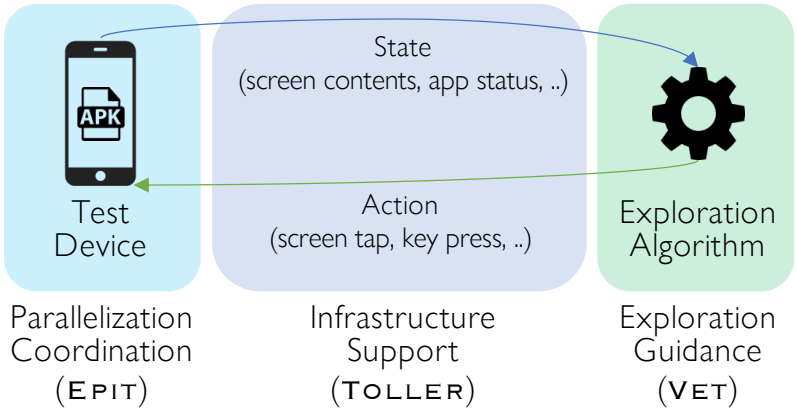


Figure 1.1: A simplified view of an automated UI test generation tool's workflow in general and this dissertation's main research work

previously proposed tools) when given a reasonable amount of test time budget.

However, subsequent measurement studies [19, 20] find that most of the existing tools barely outperform Monkey in terms of overall test effectiveness when evaluated upon comprehensive sets of Android apps, contradicting researchers' and practitioners' common belief. A study [19] in 2015 draws this conclusion based on experimental results of 10 tools (other than Monkey) on 68 relatively simple, open-source apps (where 56 out of 68 apps have fewer than 10 activities and no app has more than 50 activities). Multiple tools are proposed after this study using the 68 open-source apps as the evaluation subjects. Subsequently, our study [20] (Chapter 2) in 2018 reaffirms the conclusion based on experimental results of 5 tools (other than Monkey; 4 of them are tools not studied in [19]) on 68 popular and relatively complex industrial apps from the Google Play store [21] (where the median number of activities of apps used in our study is 50). Our study additionally shows that state-of-the-art mobile UI test generation tools generally yield low code coverage (about 30% in method coverage) after hours of testing on these popular industrial apps. The findings indicate a significant effectiveness gap that needs to be filled for automated mobile UI testing on complex industrial apps.

Aiming to understand the aforementioned effectiveness gap, we empirically investigate the test process and results from our study (Chapter 2), yielding three relevant findings. The first finding is that there is no "silver-bullet" in the scope of tools. Each tool (including Monkey) works best on a certain set of apps, as indicated by the diverse relative test effectiveness ranking among tools on different apps in our study. The fact suggests that it is difficult to build a single tool that adapts well to different apps with diverse UI designs. The second finding is that a UI test generation tool's effectiveness is not solely decided by its exploration algorithm–the tool's implementation also makes differences. For instance, we observe that some tools inject UI inputs at much lower speeds compared with Monkey in our experiments. While such implementations usually suffice on simple open-source apps, these tools will need much more time to cover various functionalities on feature-rich industrial apps, being undesirable in evaluations where the test time budget is limited.

The third finding is that it is possible to enhance the design or the implementation of different tools using unified approaches. One example is that multiple tools are found to be prone to unnecessarily repetitive UI exploration (caused by defects in the exploration algorithms or implementations), which can be revealed by automated runtime UI monitoring that does not necessarily need a tool's cooperation. In the context of existing work's focus on designing novel exploration algorithms, our findings suggest that it is worthwhile to develop complementary techniques that enhance existing tools to unleash the power of different exploration algorithms on various complex industrial apps.

Inspired by the aforementioned findings, this dissertation aims to explore the possibilities for *existing* automated UI test generation tools to achieve substantially better test effectiveness with *external* automated support (i.e., techniques that are applicable to various tools while keeping them fully automatic). This dissertation presents three parts of research on this direction, achieved by enhancing different components in the workflow of these tools (as illustrated in Figure 1.1). The first part (Chapter 3, Toller [22]) focuses on enhancing *infrastructure support* that enables a tool's exploration algorithm to obtain states from and execute actions on the test device, allowing the tool to iterate faster and cover more AUT functionalities within limited time. The second part (Chapter 4, Vet [23]) focuses on providing *exploration guidance* for a tool on a specific AUT, based on our observation that a tool's exploration algorithm or implementation might have applicability issues in certain conditions. The third part (Chapter 5, Epit) focuses on *parallelization coordination* for a tool and a specific AUT on multiple test devices to improve the overall test effectiveness or reduce testing costs by reducing overlapped explorations. Besides enhancing automated UI testing, I have also worked on UI capture/replay [24], UI regression testing [1], Android privacy [25, 26], and machine translation testing [27, 28]. However, this dissertation will focus on the work related to enhancing automated UI testing.

1.1 THESIS STATEMENT

The thesis statement of this dissertation is the following:

> Existing mobile UI test generation tools can be complemented with external automated support to achieve substantially better test effectiveness on complex mobile apps given limited test time.

In this dissertation, we first explore the opportunities of providing better infrastructure support for tools. Specifically, we find that existing state-of-the-art Android UI test generation tools spend on average 70% of their test time budget obtaining UI information from the screen (*UI Hierarchy Capturing*) and executing UI actions (*UI Event Execution*) in the testing process. Given limited test time, if we are able to improve the efficiency of these two types of UI operations, the tool can iterate faster and cover more AUT functionalities, resulting in better test effectiveness. Based

on this finding, we propose Toller, a tool consisting of much more efficient mechanism for UI Hierarchy Capturing and UI Event Execution through infrastructure enhancements to the Android operating system. After integration with existing state-of-the-art automated Android UI test generation tools, Toller substantially improves these tools' test effectiveness.

We then investigate how a tool can be guided to better explore the AUT. We specifically target *exploration tarpits*, where tools get stuck with a small fraction of app functionalities for an extensive amount of time. For example, a tool logs out an app at early stages without being able to log back in, and since then the tool gets stuck with exploring the app's pre-login functionalities (i.e., exploration tarpits) instead of its main functionalities. While tool vendors/users can manually hardcode rules for the tools to avoid specific exploration tarpits, these rules can hardly generalize, being fragile in face of diverted testing environments, fast app iterations, and the demand of batch testing product lines. We propose Vet, a general approach and its supporting system to automatically identify and resolve exploration tarpits using trace analysis and app manipulation. We find that Vet identifies exploration tarpits that reveal not only limitations in UI exploration strategies but also defects in tool implementations. Vet automatically addresses the identified exploration tarpits, enabling each evaluated tool to achieve best test effectiveness.

Last, we explore the possibility of automatically coordinating a tool's exploration on multiple devices. We mainly target the situation of deploying automated UI testing on testing clouds where developers/testers are billed by the total amount of machine time used, which brings convenience and economical benefits to mobile app developers/testers. We specifically aim to address overlapped app explorations that waste testing budgets and lead to lower overall test effectiveness. We propose Epit, a parallel testing approach that automatically manipulates the AUT on multiple devices to guide the UI test generation tool into different loosely coupled UI subspaces for exploration. We find that Epit helps substantially reduce the overlapped explorations, allowing state-of-the-art tools to reach comparable code coverage using substantially less machine time compared with the baseline.

1.2 CONTRIBUTIONS

This dissertation makes the following main contributions:

- This dissertation presents an empirical study which aims to understand the effectiveness of Android UI test generation tools on industrial apps. We directly compare the state-of-the-art tools with regard to code coverage and fault-detection ability on 68 carefully selected industrial apps, which are substantially more complex and impactful compared with open-source apps involved in previous studies. Our results suggest that there exists a significant effectiveness gap that needs to be filled for automated mobile UI testing on complex industrial apps. Additionally, our empirical investigation into the test process and results provides

insights on further improving the test effectiveness, motivating the line of work presented by this dissertation.

- This dissertation describes Toller, a tool consisting of efficient mechanism for two types of UI operations (UI Hierarchy Capturing and UI Event Execution) through infrastructure enhancements to the Android operating system. Toller injects itself into the same virtual machine as the app under test, giving Toller direct access to the app's runtime memory. Toller is thus able to directly (1) access UI data structures, and thus capture contents on the screen without the overhead of invoking the Android framework services or remote procedure calls (RPCs), and (2) invoke UI event handlers without needing to execute the UI events. Compared with the often-used UIAutomator [29], Toller reduces average time usage of UI Hierarchy Capturing and UI Event Execution operations by up to 97% and 95%, respectively. We integrate Toller with existing state-of-the-art Android UI test generation tools and achieve the range of 11.8% to 70.1% relative code coverage improvement on average. We also find that Toller-enhanced tools are able to trigger 1.4x to 3.6x distinct crashes compared with their original versions without Toller enhancement. These improvements are so substantial that they also change the relative competitiveness of the tools under empirical comparison.

- This dissertation presents Vet, a general approach and its supporting system to automatically identify and resolve exploration tarpits for the given specific Android UI test generation tool on the given specific AUT. Vet runs the tool on the AUT for some time and records UI traces, based on which Vet identifies exploration tarpits by recognizing their patterns in the UI traces. Vet then pinpoints the actions (e.g., clicking logout) or the screens that lead to or exhibit exploration tarpits. In subsequent test runs, Vet guides the test generation tool to prevent or recover from exploration tarpits. From our evaluation with state-of-the-art Android UI test generation tools on popular industrial apps, Vet identifies exploration tarpits that cost up to 98.6% test time budget. These exploration tarpits reveal not only limitations in UI exploration strategies but also defects in tool implementations. Vet automatically addresses the identified exploration tarpits, enabling each evaluated tool to achieve higher code coverage and improve crash-triggering capabilities.

- This dissertation describes Epit, a parallel testing approach that automatically manipulates the AUT on multiple devices to guide the UI test generation tool into different loosely coupled UI subspaces for exploration, to improve the overall test effectiveness or reduce testing costs by reducing overlapped explorations. Epit conducts our novel on-the-fly trace analysis during the testing process to find loosely coupled AUT UI subspaces desirable for partitioning. By controlling access to these UI subspaces during testing, Epit conceptually transforms the AUT into different variants suitable to be tested independently by the tool

on different devices. Our evaluation results show that EPIT helps state-of-the-art tools reach comparable code coverage using up to 85% less machine time than the baseline. In addition, EPIT helps reduce the overlapped explorations by up to 71.1%.

1.3 DISSERTATION ORGANIZATION

The main parts of this dissertation are organized as follows:

Chapter 2. Understanding The Effectiveness of Android UI Test Generation Tools on Industrial Apps. This chapter presents a motivational study for this dissertation's work, where we find (1) a significant effectiveness gap that needs to be filled for automated mobile UI testing on complex industrial apps, and (2) the insights that inspire this dissertation's work to bridge the gap through enhancing *existing* tools with tool-independent *external* support.

Chapter 3. TOLLER: Enhancing Infrastructure Support by System Design. This chapter presents TOLLER, a tool to provide infrastructure enhancements for UI Hierarchy Capturing and UI Event Execution to Android UI test generation tools.

Chapter 4. VET: Providing Exploration Guidance via Identifying and Avoiding UI Exploration Tarpits. This chapter presents VET, a general approach and its supporting system to automatically identify and resolve exploration tarpits for the given specific Android UI test generation tool on the given specific AUT.

Chapter 5. EPIT: Facilitating Parallelization Coordination with UI Exploration Space Partitioning. This chapter presents EPIT, a parallel testing approach that automatically manipulates the AUT on multiple devices to guide the UI test generation tool into different loosely coupled UI subspaces for effective exploration.

Chapter 6. Related Work. This chapter presents an overview of related work on automated UI testing and other relevant directions.

Chapter 7. Conclusions and Future Work. This chapter concludes the dissertation and discusses potential directions to further stretch the work from this dissertation.

CHAPTER 2: UNDERSTANDING THE EFFECTIVENESS OF ANDROID UI TEST GENERATION TOOLS ON INDUSTRIAL APPS

2.1 OVERVIEW

Researchers have also proposed various test generation tools to automate Android UI testing [8, 9, 11, 13, 14, 16, 18, 24, 30, 31, 32, 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 2.1 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. [19] 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. '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 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^3E [16], which is evaluated on only code coverage without on fault detection (note that no tool comparison is conducted there).

As shown in Table 2.1, 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 chapter, 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 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

Table 2.1: Overview of existing Android test generation tools and their evaluation subjects. ♣ indicates that both code coverage and fault detection are compared across tools on open source apps. ◊ indicates that only code coverage is measured whereas fault detection is not measured on industrial apps. △ indicates that the tool is compared with only Monkey but no other tools.

Tool/Study	Venue	Emp. Comp.		Case
		#Ind.	#Opn.	#Ind.
A^{3}E [16]	OOPSLA'13	0	0	◊28
ACTEve [18]	FSE'12	0	5	0
DroidBot [14]	ICSE-C'17	0	2	0
Dynodroid [9]	FSE'13	0	50	1000
GUIRipper [30]	ASE'12	0	♣1	0
Monkey [7]	-	-	-	-
Sapienz [8]	ISSTA'16	0	♣68	1000
Stoat [13]	FSE'17	0	♣93	1661
SwiftHand [11]	OOPSLA'13	0	10	0
WCTester [32]	FSE-Ind'16	△1	0	0
Study by [19]	ASE'15	0	♣68	0

categories and each of these apps has at least 1 million installs according to Google Play [21]. 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 chapter, 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. Overall, different tools achieve the best test effectiveness on various apps, i.e., there is no "silver-bullet" tool that outperforms all other tools on every app.

To gain better understanding of the tool performance on industrial apps, similar to a previous study methodology [34], 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 or fault detection than applying these tools individually.

In summary, this chapter makes the following main contributions:

- Empirical investigation on the effectiveness of existing available Android UI test generation tools when being applied on industrial apps. We find that there is no "silver-bullet" tool that outperforms all other tools on each app in terms of test effectiveness.
- 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 test generation tools on complex industrial apps. Our observations suggest the opportunities of achieving better test effectiveness for automated UI testing through enhancing *existing* tools with tool-independent *external* support.
- 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.

2.2 BACKGROUND

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

2.2.1 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 [35].

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.

2.2.2 Android OS Architecture

The Android OS is an open-source Linux-based software stack [36]. Android apps run within individual sandboxes, namely instances of the Dalvik Virtual Machine (DalvikVM) [37], 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 [38] 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 [39] to mitigate this limitation.

Table 2.2: Overview of Android test generation tools under study. Note that instrumentation is optional for Sapienz and Stoat.

Tool	Open Source	No Need of Modification		Exploration Strategy	No Need of App Source Code
		App	Platform		
Monkey [7]	✓	✓	✓	Random	✓
WCTester [32, 33]	✗	✓	✓	Random	✓
Sapienz [8]	✓	✓*	✗	Evolutionary	✓
Stoat [13]	✓	✓*	✓	Model-based Evolutionary	✓
DroidBot [14]	✓	✓	✓	Model-based	✓
A^3E-Depth-First [16]	✓	✗	✓	Systematic	✓

2.3 SELECTION OF ANDROID TEST GENERATION TOOLS

We choose 6 state-of-the-art UI test generation tools for our study. Monkey [7] 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 2.2 presents an overview of the test generation tools that we examine and our decision on tool selection.

2.3.1 Selected Tools under Study

Monkey Monkey [7] 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) [19].

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 [40].

Sapienz Sapienz [8, 41] is an evolutionary-testing-based test generation tool for Android UI testing. It leverages a genetic algorithm [42] 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.

Stoat Stoat [13] 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 [43] 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.

DroidBot DroidBot [14] is a programmable, light-weight, and model-based Android UI test generation 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 [44] at the time of writing.

A^{3}E-Depth-First A^{3}E [16] includes a systematic test generation tool (i.e., A^{3}E-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^{3}E. The strategy is based on high-level control flow graphs capturing activity transitions and constructed by performing static dataflow analysis on apps' bytecode.

2.3.2 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.

Dynodroid Dynodroid [9] is a guided random test generation 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.

GUIRipper GUIRipper [30] is a model-based test generation 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.

SwiftHand SwiftHand [11] is a model-based test generation 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 $\mathcal{L}^*$ [45], 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.

ACTEve ACTEve [18] is a concolic-testing [46] 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.

2.4 STUDY METHODOLOGY

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

Table 2.3: 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	✓	✓	✓	✓	✓	✓

2.4.1 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 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 2.3 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 2.4.2). '#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. As shown in Figure 2.1, on the open-source apps involved in [19], the median numbers of activities and methods are 4 and 212, respectively. On the industrial apps used in our study, the median numbers of activities and methods are 50 and 52,677, respectively.

2.4.2 Coverage/Crash Measurement

For code-coverage measurement, we use Ella [47] 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.

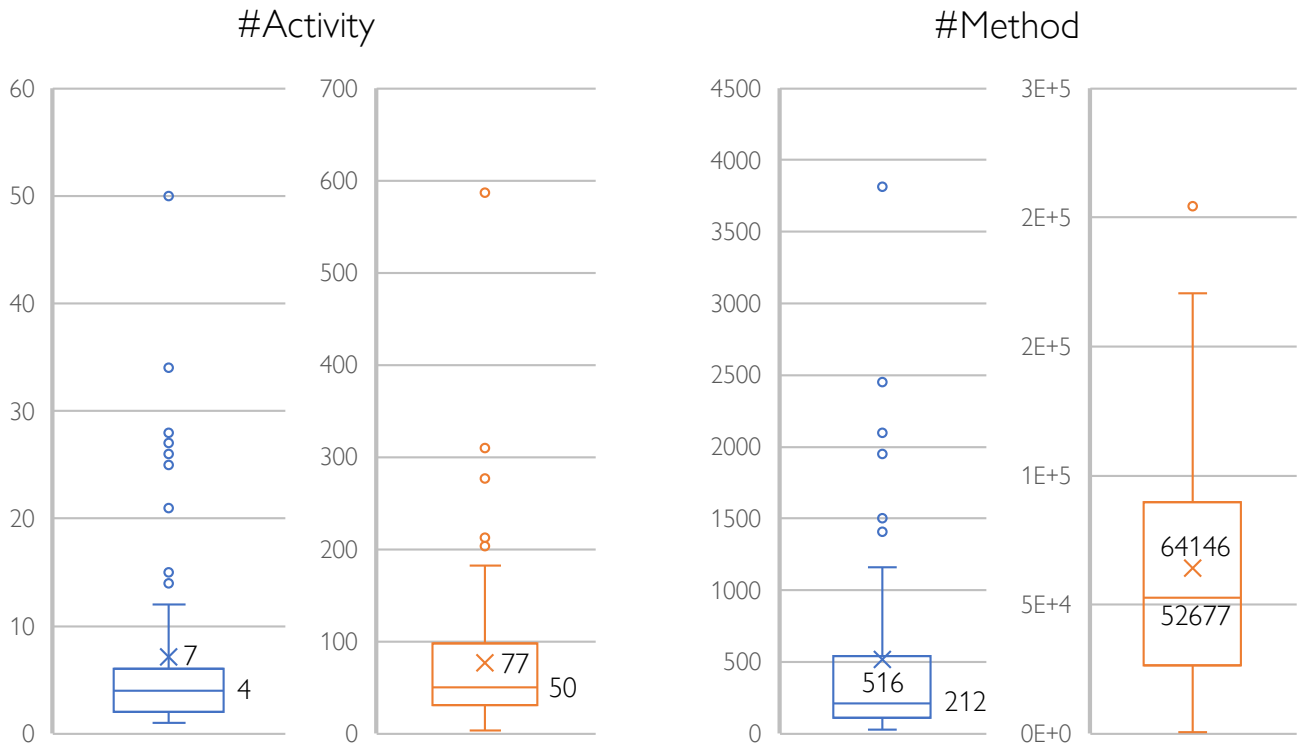


Figure 2.1: Statistics of activity and method[1] counts of open-source apps used by [19] (in blue color) and industrial apps used in our study (in orange color). Note that median and mean values are labeled in each box plot.

In practice, we find that Ella fails to instrument some large industrial apps under study due to the *64K reference limitation* of DalvikVM, 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 2.3 also shows whether each app is actually instrumented in the experiments as indicated by the '#Method' column. In total, Ella manages to instrument 41 apps in our study. 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 [48] 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).

[1]The statistics of #methods on industrial apps used in our study are obtained on 41 apps that Ella is able to instrument (see Section 2.4.2).

2.4.3 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 [13] 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).

2.5 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 test generation tool on industrial apps in our experiments.

Table 2.4: 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

Table 2.4 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 2.4, 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 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

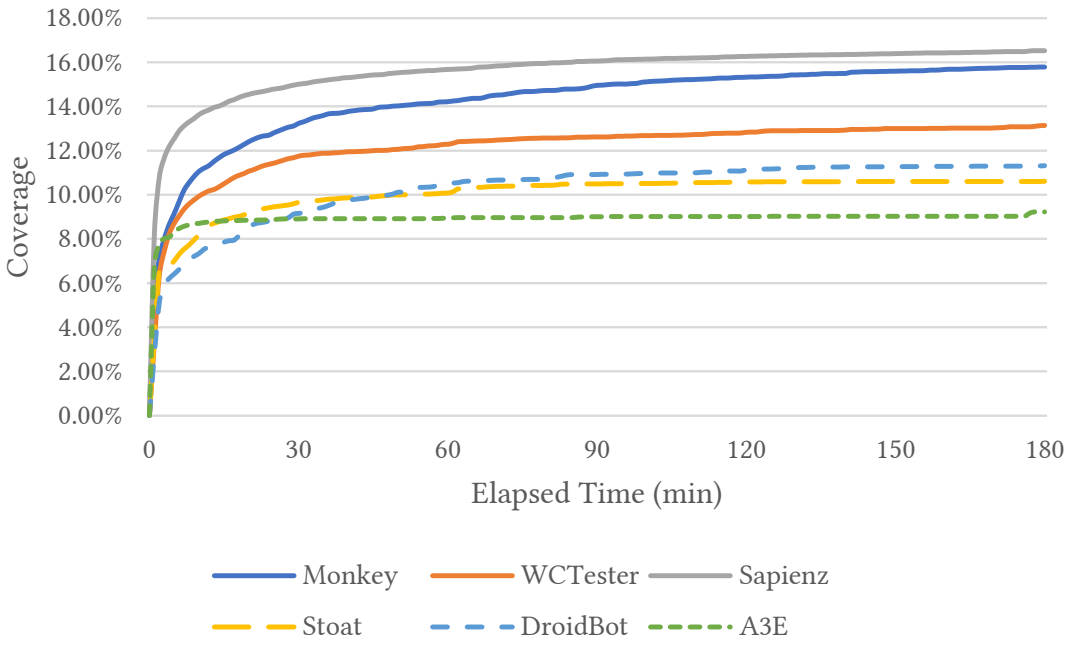


Figure 2.2: Trend of average method coverage of industrial apps achieved by test generation tools under study

over time during testing. Figures 2.2 and 2.3 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 2.2, 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 2.3, 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 2.8 for more discussion).

2.6 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 test generation tool on industrial apps in our experiments.

Table 2.4 shows the number of unique crashes triggered by each test generation tool on each

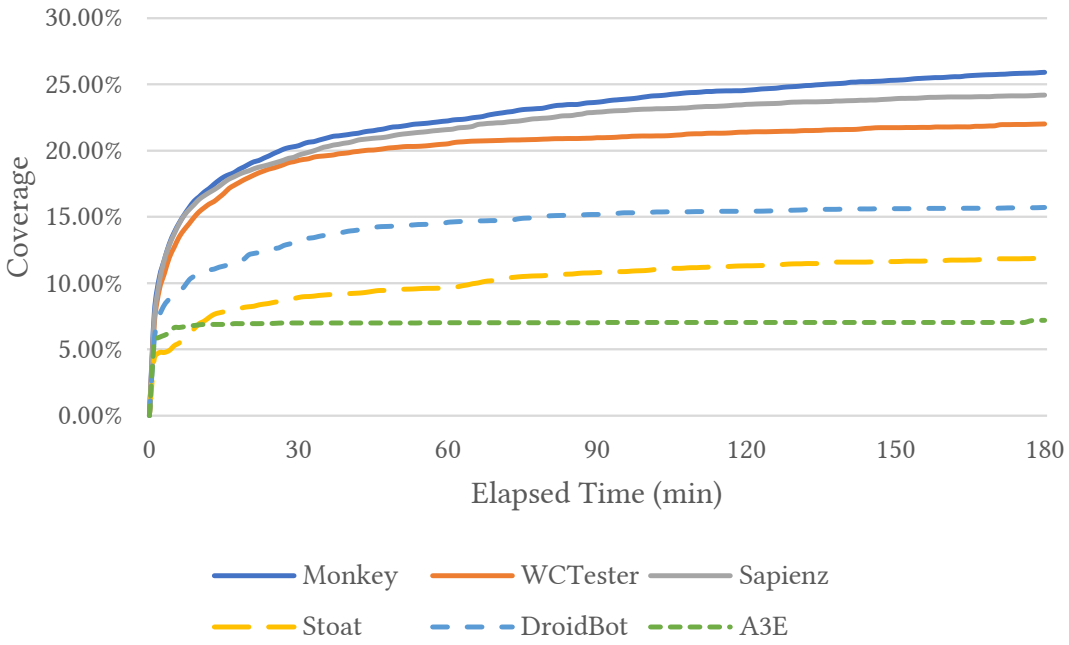


Figure 2.3: Trend of average activity coverage of industrial apps achieved by test generation tools under study

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 2.4, 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 `NullPointerException`s during starting of activities that take an `Intent` (see Section 2.2.1 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 `SQLiteException`s 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.

2.7 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 [34]. 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 2.5 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.

Table 2.5: 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

As shown in Table 2.5, 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 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 or Stoat can be good complements when Monkey is not able to cover most activities. For fault detection, combining Stoat with Sapienz 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 or Monkey can be good complements. Such suggestion is consistent with the results of manual investigation from Section 2.6, 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.

2.8 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 [47] and the Android Framework (illustrated in Section 2.2).

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 [49] shell interface. We spend no effort on setting up Monkey for industrial apps under study.

WCTester. Due to defects in the UIAutomator Python wrapper [50] 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.

We additionally monitor and investigate the testing process of a sample of test runs involving different tools with human efforts. One interesting finding is that the test effectiveness of multiple tools might be handicapped by the implementations of these tools. For instance, Stoat injects UI events at a much lower speed compared with Monkey and Sapienz in our experiments. While this implementation usually suffices on simple open-source apps when given abundant test time budget, it will require much more time to cover many functionalities on feature-rich industrial apps used in our study. The other interesting finding is that tools sometimes choose to explore UIs in highly ineffective ways, and such cases can be easily identified through monitoring. For example, Monkey and WCTester are found to get stuck on certain screens for an extensive amount of time, and such cases can be discovered by monitoring the changes of UIs during testing.

While the tool issues revealed by our aforementioned findings can be addressed by improving each tool's individual design and implementation, we find that these issues share similar mitigation strategies across different tools. One example is that multiple research tools suffer from slow

implementations, likely caused by the same inefficient UIAutomator framework used to interact with the test devices. The framework is independent of a tool's testing strategy and can be separately enhanced. Another example is that multiple tools are found to be prone to ineffective UI exploration, which can be revealed by automated runtime UI monitoring that does not necessarily need a tool's cooperation. These insights suggest the opportunities of achieving better test effectiveness for automated UI testing through enhancing *existing* tools with tool-independent *external* support.

2.8.2 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.

2.9 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 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 [51] 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.

2.10 SUMMARY

In this chapter, 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. 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. We also find that different tools achieve the best test effectiveness on various apps, i.e., there is no "silver-bullet" tool that outperforms all other tools on all apps. 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. Additionally, our observations suggest the opportunities of achieving better test effectiveness for automated UI testing through enhancing *existing* tools with tool-independent *external* support.

CHAPTER 3: TOLLER: ENHANCING INFRASTRUCTURE SUPPORT BY SYSTEM DESIGN

3.1 OVERVIEW

Aiming to understand what makes existing tools perform poorly on popular industrial apps, in this chapter, we pay particular attention to testing infrastructure's impacts on testing effectiveness; these impacts have been constantly overlooked by prior work [8, 9, 10, 11, 12, 13, 15, 16, 17, 18], which mainly emphasizes algorithmic improvements. We first conduct a motivating study to find what types of operations from these tools use most of the run time. Our study findings show that capturing information about the contents on the screen (*UI Hierarchy Capturing*) and executing UI events (*UI Event Execution*) are the two types of operations that consume the most time. In total, these two factors use on average 70% of the entire run time budget with 34% and 36% belonging to UI Hierarchy Capturing and UI Event Execution, respectively (as shown in "Combined" in Figure 3.2). These two types of operations, usually provided with *infrastructure support*, are essential for most UI test generation tools to perform their duties. Yet, we find that these two types of operations are often performed inefficiently using UIAutomator [29], a component of the Android framework. For example, we find that it can take from 0.4 to 8.2 seconds on average to capture one UI hierarchy using UIAutomator (as shown in Table 3.2). Our experiments (Section 3.5.2) find that these time usages can be reduced to just tens of milliseconds with infrastructure enhancements. The findings from our study suggest that there are substantial efficiency improvements that can be achieved with infrastructure enhancements so that tools can be more effective when given the same run time.

Based on the aforementioned findings, we propose Toller, a tool to provide infrastructure enhancements for UI Hierarchy Capturing and UI Event Execution to Android UI test generation tools. By modifying the Android framework, Toller is capable of injecting itself into any target app's virtual machine and has access to the app's runtime memory. Toller can thus directly read an app's internal UI data structures and quickly extract the app's UI hierarchy, avoiding much of the overhead caused by using UIAutomator, which relies on the complicated internal logic of the Android framework as well as remote procedure calls. Toller also enables the direct invocation of UI event handlers, thereby eliminating the unnecessary time spent on executing low-level UI events that simulate human interactions (e.g., waiting for long clicks) and have to be translated to UI element-specific events based on the UI hierarchy. Our experiments show that Toller can substantially reduce the time required for the aforementioned two types of operations.

We integrate Toller with three state-of-the-art Android UI test generation tools that depend on UIAutomator: *Stoat* [13], *WCTester* [32, 33], and a tool named *Chimp*, which we implement following a similar algorithmic design as the original *Monkey* [7]. Our experiments with 15 popular, industrial apps obtained from the Google Play Store show that the average time usages for UI

Hierarchy Capturing are reduced by about 97%, 77%, and 97% on Chimp, WCTester, and Stoat, respectively, and for UI Event Execution, the average time usages are reduced by 40%, 18%, and 95% on Chimp, WCTester, and Stoat, respectively.

Given the same run time for tools with and without Toller, the Toller-enhanced tools are able to execute more events than the original versions of the tools, and the Toller-enhanced tools on average achieve higher code coverage and trigger more distinct crashes than the original versions of the tools. In fact, our experiments show that Toller-enhanced tools achieve 11.8% to 70.1% relative code coverage improvement on average and are able to trigger 1.4x to 3.6x distinct crashes compared with the original versions of the tools. These improvements are so substantial that they even change the relative competitiveness of the tools under empirical comparison. For instance, Stoat achieves a higher average code coverage compared to Monkey only after Stoat is enhanced with Toller. We additionally involve another Android UI test generation tool, Ape [15], in our experiments. We find that Ape is already benefiting from its own improved infrastructure support, despite the fact that the tool authors did not mention the improved infrastructure support in their paper. We make Toller's source code and the scripts used to set up Toller publicly available [52]. We hope that our results can raise the community's awareness of the significance of infrastructure support beyond the community's existing heavy focus on algorithms.

This chapter makes the following main contributions:

- A motivating study to understand what types of operations use most of existing Android UI test generation tools' run time. Our results show the potential of two infrastructure enhancements for improving these tools' testing efficiency.
- Design and implementation of Toller, which provides two infrastructure enhancements to Android UI test generation tools so that they can benefit from efficient UI Hierarchy Capturing and UI Event Execution support.
- Comprehensive experiments involving the integration of Toller with Android UI test generation tools. Our experiments show that infrastructure enhancements can lead to substantial effectiveness improvements.

3.2 BACKGROUND

This section presents the relevant background information of Toller, including two Android UI system interfaces used mainly by test generation tools, the structure of the Android framework, and UIAutomator.

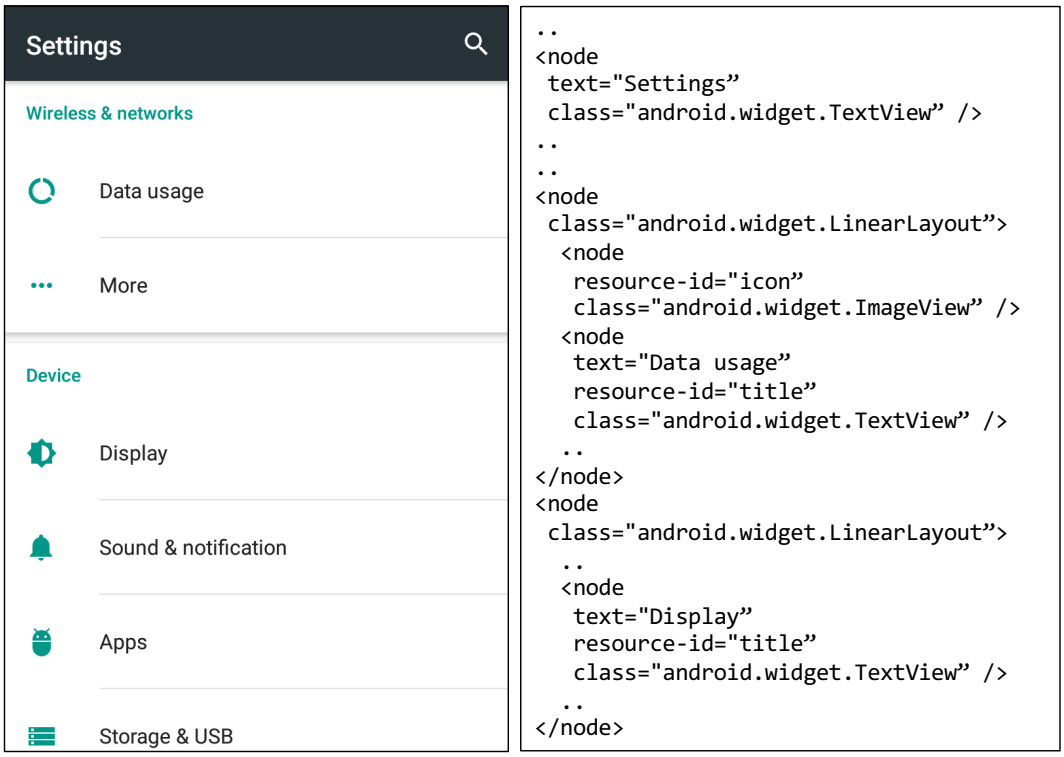


Figure 3.1: A simplified example of captured UI hierarchy

3.2.1 Android System Interfaces for Testing Tools

This section introduces the two Android UI system interfaces that are used by most test generation tools and that Toller aims to tackle: *UI Hierarchy Capturing* and *UI Event Execution*.

UI Hierarchy Capturing enables UI test generation tools to obtain detailed information about current on-screen contents, including UI properties (e.g., widget type, location, and size) and hierarchical settings (e.g., some widget being a child of another widget). The captured UI information serves as context for testing decision making and is especially critical to model-based tools.

Figure 3.1 shows a simple example of a captured UI hierarchy (represented in XML format) along with its corresponding screenshot. Each `node` in the hierarchy depicts a `View` (abstraction of UI elements), which can be either a `ViewGroup` (`Views` specifically for holding and organizing other `Views`) or ordinary `View` (i.e., UI elements) that users can see and interact with. The hierarchical relations among `Views` are reflected by "child of" relations of `nodes`. Each `node` contains UI properties (e.g., text, position, resource ID) that vary across different types and instances of UI elements.

UI Event Execution enables tools to perform UI events (e.g., screen clicks, text inputs) on the app under test. The interface is usually invoked after each UI Hierarchy Capturing, where at each step, a tool gets the current UI state and then executes a UI event based on the state of the UI. A common way of performing UI events is to inject the corresponding *low-level UI events* into

the Android system. For example, long clicking some point (x, y) on the screen can be decomposed into (1) touching down at (x, y), (2) waiting for 0.5 seconds, and (3) touching up at (x, y). These actions are processed as if they were from human users.

3.2.2 Structure of Android Framework

This section introduces how app bytecode runs on Android, as well as how the Android framework is structured. This section helps explain how Toller is integrated with the Android framework.

The Android framework can be divided into two parts: the *app space* part, which runs in the same virtual machine (VM) as each app's bytecode, and the *system service* part, which runs in standalone VMs and communicates with the app space part through RPCs. The app space part of the Android framework consists of a number of fundamental Java classes that are accessible from every Android app. These fundamental classes are preloaded into the VM and cannot be overridden by app classes. These characteristics of the app space part make it ideal to host Toller's runtime stub, which needs to gain direct access to each app's runtime memory. Section 3.4 presents more details about how Toller makes use of direct access to app runtime memory and the benefits of doing so.

3.2.3 UIAutomator

One component of the Android framework is UIAutomator [29], the standard service for UI interactions on an Android device, used by not only automated UI test generation tools but also UI test scripting platforms such as Espresso [53].

Implementation of UIAutomator can be divided into three parts. The first part runs as a system service, which coordinates all UIAutomator related activities on the device. The second part resides in the app space Android framework, responsible for collecting UI-related information from the app runtime memory and communicating with the system service counterpart. The third part acts as a client to the system service, with which a user can request UI information to be captured or UI event to be executed.

There is much overhead in using UIAutomator, especially when it is used to capture UI hierarchies. When a user sends a request to the system service for UI Hierarchy Capturing through remote procedure calls (RPCs), the service first needs to look up the active UI windows and then dispatch the request using RPCs to each app process owning the UI windows. When the app space Android framework counterpart receives the request, it uses accessibility interfaces to gather the UI hierarchy for each requested UI window and transmits the UI hierarchies back to the system service. When each app process has finished processing, the system service finally formats the UI hierarchies into one XML document and then transmits the document back to the user. Section

3.4 presents details for how Toller can reduce this overhead.

3.3 MOTIVATING STUDY

A recent study [20] has found that Android test generation tools are substantially less effective on popular industrial apps, compared with open-source apps, which are often used for evaluation. Many popular industrial apps are feature-rich and have much larger codebases than open-source apps. Existing work [8, 9, 10, 11, 12, 13, 15, 16, 17, 18] has been focusing on designing sophisticated UI exploration algorithms to achieve better testing effectiveness. Although testing effectiveness can be improved with sophisticated algorithms, one often overlooked aspect is the efficiency of infrastructure support, which is necessary for tools to perform their duties. Given that a common way of evaluating Android test generation tools is to set a run time limit and measure code coverage or crash triggering ability at the end of the run time, the efficiency of infrastructure support directly affects a tool's overall testing effectiveness.

To guide enhancements to Android test generation tools, we conduct a motivating study to understand the extent and sources of inefficiency from infrastructure support. Our study focuses on understanding the (in)efficiencies of Android UI test generation tools interacting with the testing devices. Our findings enable us to design and implement a general solution for different tools. While UI Hierarchy Capturing and UI Event Execution are necessary parts of tool-device interactions, test generation tools can also have other types of interactions. For example, a tool may execute a shell command through Android Debug Bridge (ADB) [49] to start the target app. Our motivating study aims to understand the time usages by different types of interactions to learn about their potentials for enhancements.

3.3.1 Experiment Settings

To drive our design of Toller, we run and profile three tools: Chimp, WCTester [33], and Stoat [13]. All three of these tools use UIAutomator [29] and control the testing devices from a computer (i.e., having no on-device components themselves), making it easy to profile the tools' interactions with the testing devices. The following are more details about these three tools:

- We implement *Chimp*, a tool based on Monkey [7]. Similar to Monkey, Chimp randomly decides on what UI events to generate. The main difference between Chimp and Monkey is that Chimp is aware of the UI element locations when generating UI events while Monkey is unaware. Note that we do not include Monkey directly in our study because it does not capture any UI information from the target app and just injects random low-level UI events. Therefore, the enhancements provided by Toller are unlikely to improve Monkey's testing effectiveness. On the other hand, Chimp shares the same exploration strategy as Monkey

Table 3.1: Overview of industrial apps for experiments. Note that '#Inst' denotes the approximate number of downloads.

App Name	Version	Category	#Inst	APK Size
Abs	4.2.0	Health & Fitness	10m+	57 MB
Duolingo	3.75.1	Education	100m+	12 MB
Filters For Selfie	1.0.0	Beauty	1m+	21 MB
GoodRx	5.3.6	Medical	1m+	12 MB
Google Translate	6.5.0.RC04	Tools	500m+	26 MB
Marvel Comics	3.10.3	Comics	5m+	6.2 MB
Merriam-Webster	4.1.2	Books & Reference	10m+	66 MB
Mirror	30	Beauty	1m+	3.3 MB
My baby Piano	2.22.2614	Parenting	5m+	3.7 MB
Sketch	8.0.A.0.2	Art & Design	50m+	25 MB
trivago	4.9.4	Travel & Local	10m+	12 MB
WEBTOON	2.4.3	Comics	10m+	23 MB
Word	16.0.9126	Productivity	100m+	74 MB
YouTube	15.35.42	Video Player & Editor	1b+	93 MB
Zedge	7.2.2	Personalization	100m+	33 MB

and makes use of Toller's infrastructure enhancements. Specifically, at each step, Chimp (1) obtains the current UI hierarchy, (2) determines what UI event types are executable (i.e., there is at least one UI element with a corresponding action handler) on the current screen, (3) randomly chooses a UI event type based on a predefined probability distribution, and (4) randomly chooses an applicable UI element (and action parameters if needed) to apply the next action on.

- *WCTester* [33] is a practical upgrade from Monkey, featuring widget awareness, state awareness, and various heuristics to improve its effectiveness. Developed by researchers and practitioners [32, 33], the tool has been deployed on WeChat, an app with over one billion monthly active users. The tool has moderate testing effectiveness on various industrial apps according to a previous study [20]. Although WCTester is not open-sourced, the authors [32, 33] shared the tool with us upon request.

- *Stoat* [13] is a sophisticated model-based tool featuring probabilistic modeling and sampling-based model evolution. While the Stoat paper [13] reports that Stoat outperforms Monkey based on an evaluation using open-source apps, a previous study [20] shows that Stoat generally achieves low code coverage on popular industrial apps and achieves lower code coverage than Monkey. Stoat is open-sourced.

All of our experiments are conducted on the official Android x86-64 emulators running Android 6.0 on a server with Xeon E5-2650 v4 processors. Each emulator is allocated with 4 dedicated CPU cores, 2 GiB of RAM, and 2 GiB of internal storage space. The emulators are stored on a

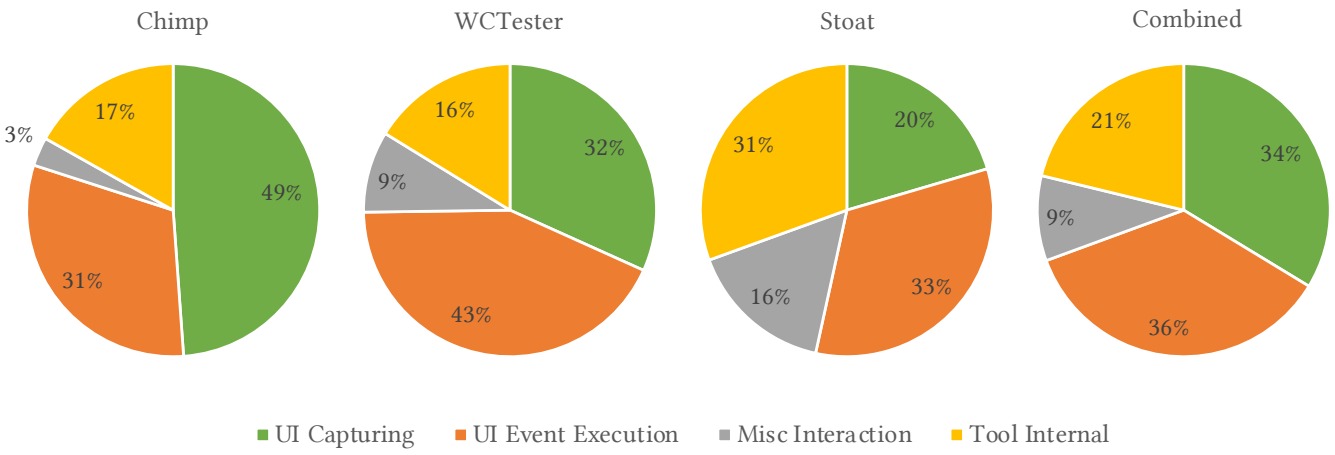


Figure 3.2: Time usage distribution by operation types

RAM disk and backed by discrete graphics cards for minimal mutual influences caused by disk I/O bottlenecks and CPU-intensive graphical rendering.

Our experiments consist of 15 widely used industrial apps from the Google Play Store and are selected from obtaining the top apps from 13 different categories. The selected apps must all run on Android 6.0 x86-64 emulators and do not require logging in, given that logins can be flaky due to network calls and require expensive manual checks afterwards. Details of these apps are shown in Table 3.1. Each test generation tool runs on each app for one hour without interruption. If a tool exits before using up the run time budget in some run, we automatically launch the tool to start testing again until the allotted one hour is up.

3.3.2 Results

Figure 3.2 shows the breakdown of time usages by different types of tool operations. Specifically, we measure the number of occurrences as well as the end-to-end time usages of three types of operations: UI Hierarchy Capturing, UI Event Execution, and ADB command executions that are not used for the first two purposes (labeled as "Misc Interaction"). The remaining time used during testing is then considered to be used internally by the tool.

As shown in Figure 3.2, UI Hierarchy Capturing and UI Event Execution take most of the run time budget on all of the studied tools. By putting results from all three tools together ("Combined"), we see that these two types of interactions each take about 1/3 of the entire run time. Our findings suggest that focusing on the two types of operations has good potential for improving the efficiency of these tools and consequently the effectiveness of the tools when the tools are given a specific run time budget. In Section 3.5, we show how the use of Toller to enhance UI Hierarchy Capturing and UI Event Execution can lead to higher average code coverage and better crash triggering ability for the same three tools and 15 apps used in this motivating study.

3.4 DESIGN AND IMPLEMENTATION OF TOLLER

Figure 3.3 shows an overview of TOLLER's design. TOLLER's source code and our scripts to set up TOLLER are publicly available [52]. TOLLER resides in the same VM as the app under test, giving TOLLER fast and direct access to the app's runtime memory. TOLLER is thus able to (1) resolve the app's internal UI-related data structures to generate UI hierarchies, and (2) dynamically analyze, invoke, or alter UI event handlers to perform UI events or understand/control app behaviors. To resolve UI-related data structures, TOLLER uses Java reflection to read the single-instanced `AccessibilityManager` class that indirectly points to all visible windows' root view groups. TOLLER then recursively finds all Java objects corresponding to child views (i.e., instances of subclasses of `android.view.View`) to generate hierarchies. To invoke UI event handlers, TOLLER directly calls the corresponding action invocation methods (e.g., `performClick()`) upon `View` objects. Note that TOLLER's UI Event Execution strategy falls back to low-level UI event injections on event handlers that have not been covered by a low-level UI event injection. This strategy helps TOLLER by (1) invoking low-level UI event injections at least once for every event handler that TOLLER directly invokes and (2) preventing the direct invocation of event handlers from covering less code than low-level UI event injections. These low-level injections can cover more code than directly invoking a specific event handler, say EH, because low-level injections may first invoke a topmost `View`'s event handler only for it to then invoke a child `View`'s event handler until the event eventually reaches EH.

We also design TOLLER to be non-intrusive to the app under test: TOLLER is bundled with the app-space Android framework classes on the testing device and app installation packages are not modified. This design is particularly useful for testing close-sourced industrial apps, because (1) many apps have self-protection mechanisms, preventing unauthorized changes to the installation packages, and (2) manipulating a large app's bytecode is highly error-prone (for example, just adding a new class during instrumentation may cause an app's `.dex` file to exceed the 64K method limit [39]).

For our experiments, we integrate TOLLER with the Android 6.0 framework on both emulators and real devices. While we experience no issue with our way of integration, we would still like to point out that it is possible to use TOLLER without modifying the Android framework; in such a case, developers could simply need to add TOLLER to the app's codebase when building the app. We discuss more about the trade-offs of this option in Section 3.7. In brief, we take the following steps to inject TOLLER into an Android framework. First, we obtain the Android framework's DEX bytecode from the target device using ADB. Second, we convert the Android framework's bytecode into Smali [54] IR code. Third, we compile TOLLER's source code into Smali code. Fourth, we modify the Android framework's Smali code to incorporate TOLLER's Smali code. Finally, we convert all Smali code into DEX bytecode and replace the Android framework's bytecode on the target device with the converted DEX bytecode.

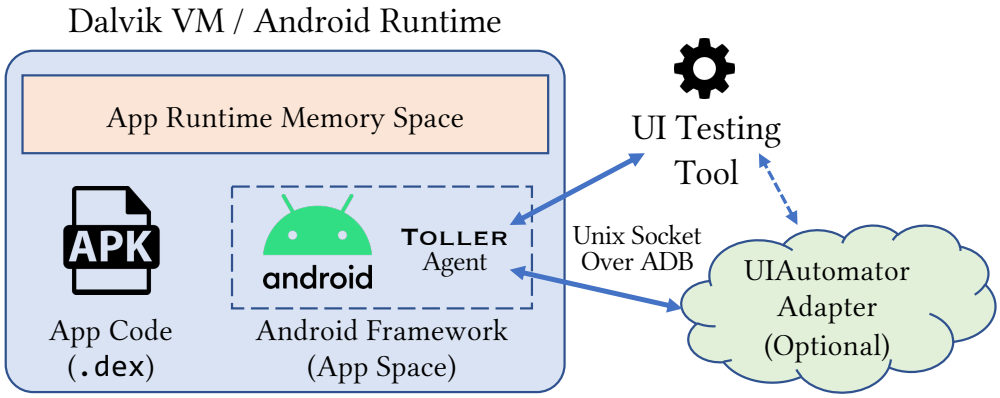


Figure 3.3: Overview of Toller's design

As Android test generation tools typically run on a computer, while the app under test and TOLLER run on a device, the tools and TOLLER need a mechanism to communicate (e.g., share UI hierarchy information) with one another. TOLLER uses Unix's abstract socket for this communication as Android already provides good support (e.g., `LocalServerSocket`) for such communication. By doing so, TOLLER also does not need the app to have any specific permissions (e.g., networking, read/write storage).

As developers often change `UncaughtExceptionHandler`s when apps are first started to avoid sharing implementation details, TOLLER can also be used to disable apps' `UncaughtExceptionHandler`s. Disabling such handlers helps ensure that stack traces are printed to system logs so that experiments can understand crash statistics from such logs (e.g., our experiments in Section 3.5.4). To disable such handlers, TOLLER periodically (every five seconds in our experiments) calls `Thread.setDefaultUncaughtExceptionHandler()` to restore the default handler to ensure that stack traces from crashes are printed to the system logs.

TOLLER's implementation of UI Event Execution relies on TOLLER's UI Hierarchy Capturing. Concretely, in a TOLLER-captured UI hierarchy, each UI element is associated with a globally unique identifier, which is linked to the memory address of the underlying `View` object (see Section 3.2.2 for more details on how UI hierarchies are represented). TOLLER-enhanced test generation tools can subsequently use these identifiers to precisely specify the UI element that should be executed. This feature is especially helpful when (1) an app's UI is constantly changing and UI events from the tools cannot be easily executed on the desired UI element, and (2) a tool wants to execute UI events on not-easily accessible UI elements such as list items that are visible only after scrolling. In general, we find that TOLLER with UI Hierarchy Capturing and UI Event Execution has better testing effectiveness than TOLLER with just UI Hierarchy Capturing (see Section 3.5.6 for more details). Therefore, in all of our experiments except Section 3.5.6, we define TOLLER-enhanced as TOLLER with both UI Hierarchy Capturing and UI Event Execution.

Table 3.2: UI Hierarchy Capturing (Capt) efficiency comparison. Note that for each tool T, T_O refers to its original version, while T_E refers to our Toller-enhanced version. Time is shown in milliseconds.

	$\textbf{Chimp}_O$	$\textbf{Chimp}_E$	$\textbf{WCTester}_O$	$\textbf{WCTester}_E$	$\textbf{Stoat}_O$	$\textbf{Stoat}_E$
Time per capt	846	22	360	82	8175	245
# of capt	31182	47599	47666	83603	1348	47525

Table 3.3: UI Event Execution (Exec) efficiency comparison. Note that for each tool T, T_O refers to its original version, while T_E refers to our Toller-enhanced version. Time is shown in milliseconds.

	$\textbf{Chimp}_O$	$\textbf{Chimp}_E$	$\textbf{WCTester}_O$	$\textbf{WCTester}_E$	$\textbf{Stoat}_O$	$\textbf{Stoat}_E$
Time per exec	762	454	455	372	8395	391
# of exec	22045	57714	51137	65340	2118	44466

3.5 EVALUATION

To understand the impact that Toller's infrastructure enhancements can have on Android test generation tools, we investigate five main research questions:

RQ1: How does each of Toller's infrastructure enhancements contribute to Android test generation tools' efficiency?

RQ2: How does enhancing Android test generation tools with Toller improve achieved code coverage?

RQ3: How does enhancing Android test generation tools with Toller improve achieved crash triggering ability?

RQ4: How much do covered code entities and triggered crashes overlap for each tool that is and is not enhanced with Toller, respectively?

RQ5: How does each of Toller's infrastructure enhancements contribute to Android test generation tools' effectiveness?

We address RQ1 to understand how Toller's various infrastructure enhancements affect the run time performance benefits of Toller. We address RQ2 and RQ3 to understand how Toller affects Android test generation tools on two metrics commonly used to evaluate such tools and to understand whether the effectiveness rankings of these tools change when they are and are not enhanced with Toller, respectively. We address RQ4 to understand the extent that a tool enhanced with Toller covers the same code entities and triggers the same crashes as the tool not enhanced with Toller. Finally, we address RQ5 to understand how Toller's various infrastructure enhancements affect two metrics commonly used to evaluate Android test generation tools.

Table 3.4: Average method coverage for all tool versions. Note that Mk, Ch, Wt, and St denote Monkey, Chimp, WCTester, and Stoat, respectively. For each tool T, $\mathbf{T}_O$ refers to its original version, while $\mathbf{T}_E$ refers to our Toller-enhanced version. $\mathbf{Ape}_S$ refers to the slow version of Ape. Each integer cell shows the average number of covered distinct methods across three runs by the corresponding tool version on the corresponding app. For each tool T, $\Delta = (\mathbf{T}_E - \mathbf{T}_O)/\mathbf{T}_O \times 100\%$. Average $\Delta = (\overline{\mathbf{T}}_E - \overline{\mathbf{T}}_O)/\overline{\mathbf{T}}_O \times 100\% = (\Sigma\mathbf{T}_E - \Sigma\mathbf{T}_O)/\Sigma\mathbf{T}_O \times 100\%$.

App Name	$\mathbf{Ape}_S$	$\mathbf{Ape}_O$	Δ	Mk	$\mathbf{Ch}_O$	$\mathbf{Ch}_E$	Δ	$\mathbf{Wt}_O$	$\mathbf{Wt}_E$	Δ	$\mathbf{St}_O$	$\mathbf{St}_E$	Δ
Abs	8273	8424	1.8%	6213	6656	6874	3.3%	7527	7622	1.3%	3538	5282	49.3%
Duolingo	14180	14598	2.9%	8948	13500	13447	-0.4%	11659	13482	15.6%	6297	14006	122.4%
Filters For Selfie	2369	5489	131.7%	4077	2290	2262	-1.2%	2608	2170	-16.8%	2345	2227	-5.0%
GoodRx	15524	14272	-8.1%	13149	14033	15499	10.4%	13829	15904	15.0%	8716	12243	40.5%
Google Translate	8918	9169	2.8%	7554	7376	8477	14.9%	8793	8960	1.9%	3653	6855	87.7%
Marvel Comics	5306	5873	10.7%	4538	4672	4781	2.3%	4378	4460	1.9%	3459	4908	41.9%
Merriam-Webster	8229	8287	0.7%	5141	7657	8486	10.8%	6661	8241	23.7%	7238	7856	8.5%
Mirror	1120	1124	0.4%	426	1253	657	-47.6%	1105	851	-23.0%	887	948	6.9%
My Baby Piano	1096	3419	212.0%	165	1583	1652	4.4%	1373	1553	13.1%	700	219	-68.7%
Sketch	7695	8124	5.6%	6871	8081	8532	5.6%	7666	7782	1.5%	5755	7766	34.9%
trivago	19491	20079	3.0%	19678	18721	19317	3.2%	19373	19164	-1.1%	4325	19424	349.1%
WEBTOON	20415	24493	20.0%	19775	12590	21457	70.4%	14329	23338	62.9%	5695	13672	140.1%
Word	12057	12387	2.7%	11911	10875	12713	16.9%	12834	12612	-1.7%	8445	11482	36.0%
Youtube	28026	24888	-11.2%	17945	17086	18123	6.1%	17923	18930	5.6%	11162	18434	65.1%
Zedge	32937	43080	30.8%	28139	35532	38786	9.2%	34530	36665	6.2%	19490	30664	57.3%
Average	**12376**	**13580**	**9.7%**	**10302**	**10794**	**12071**	**11.8%**	**10973**	**12116**	**10.4%**	**6114**	**10399**	**70.1%**

3.5.1 Evaluation Setup

To answer our RQs, we use the same experiment environment and set of apps as our motivating study (Section 3.3.1). We collect the method coverage as code coverage achieved by each run using the MiniTrace [55] tool, which modifies DalvikVM/ART and does not require app instrumentation. We consider only crashes originated from app bytecode and collect code locations in stack traces as crash signatures. We obtain stack traces by monitoring and filtering Android Logcat [48] messages. As mentioned in Section 3.4, we use Toller to remove apps' `UncaughtExceptionHandler`s to ensure that stack traces are being reported to Logcat.

In addition to the three Android UI test generation tools used in our motivating study (Section 3.3.2), we also use *Ape* [15], another state-of-the-art tool, for our experiments. We use the four tools in the following settings.

- Both WCTester and Stoat run on computers and use UIAutomator to capture UI hierarchies. To enhance the two tools with Toller while keeping implementation changes minimal, we translate Toller's captured UI hierarchies to the UIAutomator's format to make them directly readable by these two tools. For UI Event Execution, WCTester injects low-level UI events directly using ADB shell commands, while Stoat sends UI element queries to UIAutomator to generate and inject the corresponding low-level UI events. We replace both tools' original implementation of UI Event Execution with Toller.

- Like WCTester and Stoat, Chimp also runs on computers and can use UIAutomator to capture UI hierarchies. To enhance Chimp with Toller, we choose to fully incorporate

Toller into Chimp to avoid unnecessary translations of UI hierarchies. For UI Event Execution, Chimp injects low-level UI events directly using ADB shell commands while its Toller-enhanced version uses Toller for UI Event Execution.

- As the most recently proposed state-of-the-art tool in our experiments, Ape already provides support for fast UI Hierarchy Capturing in its implementation by using hidden Android accessibility service APIs. These implementation details are not explicitly discussed in the tool's paper [15]. To show the significance of infrastructure support on Ape, we modify Ape to build its slow version, which leverages UIAutomator services in the same way as the other tools. We then compare the slow version's testing effectiveness with the original version of Ape. Note that unlike the other Toller-enhanced tools, the original Ape contains only the UI Hierarchy Capturing enhancement and not the UI Event Execution enhancement because UI Event Execution requires Toller's UI Hierarchy Capturing (Section 3.4).

In total, we have nine tool versions in our experiments: Chimp (with and without Toller), WCTester (with and without Toller), Stoat (with and without Toller), Ape (original and slow version), and Monkey. To compensate for potential randomness in our experiments introduced by tool or app logic, we run each tool on each app three times, with each run being one hour. Overall, we spend 27 hours per app (9 tool versions * 3 runs for each version) and a total of 405 hours (27 * 15 apps) for all apps.

3.5.2 RQ1: Efficiency of Enhancements

To understand how Toller's UI Hierarchy Capturing and UI Event Execution infrastructure enhancements affect the run time performance benefits of Toller, we integrate Toller with the three test generation tools from Section 3.3.1 and re-run the experiments in the same settings. The time usage statistics of UI Hierarchy Capturing and UI Event Execution are shown in Tables 3.2 and 3.3, respectively. Note that the numbers for each tool are aggregated from running on all 15 apps once. As shown in Tables 3.2 and 3.3, Toller is capable of reducing overheads for both primitive interfaces on all three tools. Specifically, the average time usages for UI Hierarchy Capturing are reduced by about 97%, 77%, and 97% on Chimp, WCTester, and Stoat, respectively. For UI Event Execution, the average time usages are reduced by 40%, 18%, and 95% on Chimp, WCTester, and Stoat, respectively. We find that UI Event Execution has less substantial overhead reductions than UI Hierarchy Capturing because as described in Section 3.4, Toller falls back on using low-level UI event injections on event handlers that have not been covered by a low-level UI event injection.

UI Hierarchy Capturing and UI Event Execution can take a substantially different amount of time for different tools because some of the tools use different approaches to invoke UIAutomator (e.g., directly invoking the `uiautomator` command in the ABD shell as used by Chimp, or using

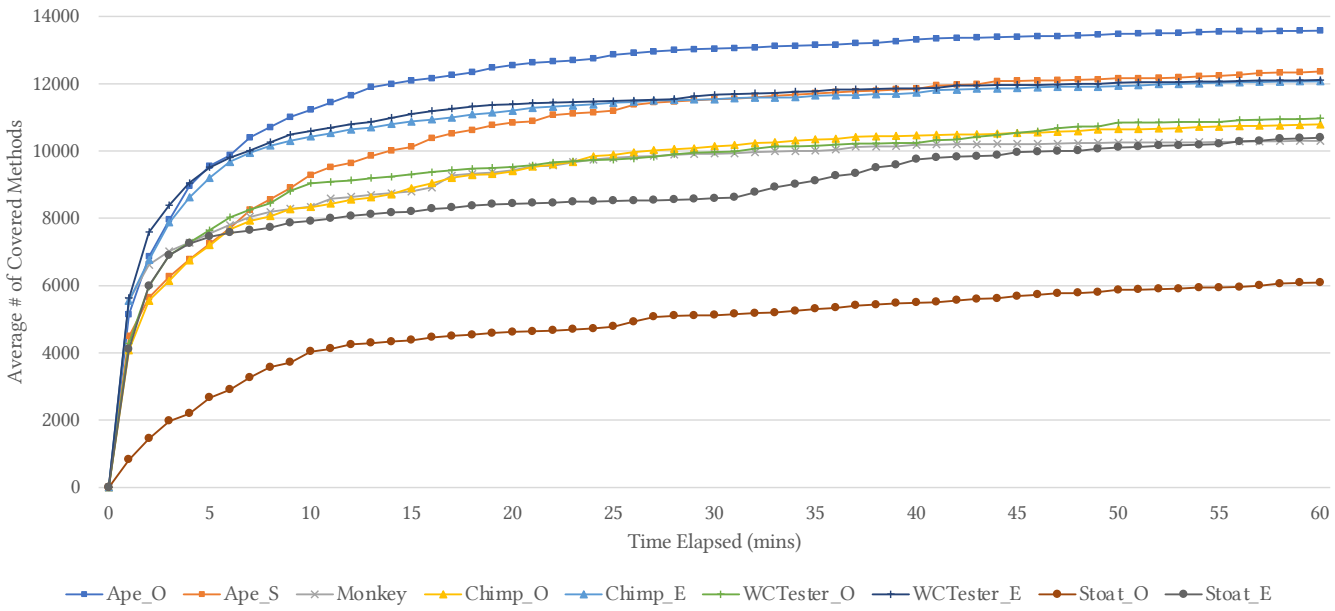


Figure 3.4: Average method coverage by elapsed time during testing. Note that each data point shows how many methods have been covered on average across three runs on all apps by the respective tool, after the corresponding amount of time has elapsed in each run. The ending number of covered methods for each tool is the same as that in the "Average" row in Table 3.4.

a service wrapper [29] for ease of programming in the case of WCTester and Stoat). Another observation is that the original implementation of Stoat takes much more time than the other tools to perform both types of operations. We find that this result is related to how Stoat uses the UIAutomator service wrapper: Stoat's implementation essentially sets up and establishes new connections to the on-device service agent before *each* capture or action. On the contrary, WCTester sets up this connection only once and persists the connection, eliminating much overhead.

3.5.3 RQ2: Code Coverage Benefits

Table 3.4 shows the average coverage statistics of each pair of tools and apps from our experiments. Figure 3.4 shows the changes of average code coverage across all apps for each tool along with run time. Note that methods that can be covered after app launch but before testing starts are excluded. As shown in Table 3.4 and Figure 3.4, using Toller's infrastructure enhancements helps improve the testing effectiveness of various Android UI test generation tools. Specifically, enhancing Chimp, WCTester, and Stoat with Toller yields 11.8%, 10.4%, and 70.1% average method coverage improvements, respectively. For Ape, the tool's own fast UI Hierarchy Capturing implementation brings 9.7% average method coverage improvement. It should also be noted that the aforementioned percentages are calculated based on the average number of covered methods across different apps, where apps with a larger codebase can have a bigger impact on the results.

One key finding is that the differences of code coverage brought by infrastructure enhancements

Table 3.5: Cumulative numbers of distinct crashes for all tool versions. Note that Ch, Wt, and St denote Chimp, WCTester, and Stoat, respectively. For each tool T, T_O refers to its original version, while T_E refers to our Toller-enhanced version. Each integer cell under T_O or T_E shows the cumulative number of distinct crashes across three runs by the corresponding tool version on the corresponding app. A blank cell indicates no crash. Each integer cell under ΣT indicates the union number of covered methods by the two tool versions on the corresponding app. $\%_O = T_O/\Sigma T \times 100\%$, similar for $\%_E$. In the 'Total' row, each integer indicates the sum value of all numbers in the respective column, while each percentage is calculated from sum values using the same methodology as aforementioned.

App Name	Ape_S	$\%_S$	Ape_O	$\%_O$	ΣApe	MK	Ch_O	$\%_O$	Ch_E	$\%_E$	ΣCh	Wt_O	$\%_O$	Wt_E	$\%_E$	ΣWt	St_O	$\%_O$	St_E	$\%_E$	ΣSt
Abs	1	33%	2	67%	3	3	1	100%			1	3	75%	1	25%	4	8	67%	12	100%	12
Duolingo	1	50%	1	50%	2			-		-				1	100%	1	6	55%	9	82%	11
Filters For Selfie		-		-		1		-		-			-		-		3	75%	4	100%	4
GoodRx			1	100%	1				5	100%	5	1	13%	7	88%	8	6	67%	5	56%	9
Google Translate			1	100%	1				1	100%	1		-		-		8	73%	5	45%	11
Marvel Comics	1	100%			1				1	100%	1			1	100%	1	9	82%	9	82%	11
Merriam-Webster		-		-				-		-			-		-		4	44%	9	100%	9
Mirror	3	60%	5	100%	5	5	3	60%	5	100%	5	5	83%	4	67%	6	5	63%	7	88%	8
My Baby Piano		-		-				-		-			-		-			-		-	
Sketch		-		-				-		-			-		-		4	80%	4	80%	5
trivago	1	33%	2	67%	3	3	1	100%			1			1	100%	1	8	53%	11	73%	15
WEBTOON			1	100%	1	1		-		-		1	100%			1	8	57%	14	100%	14
Word			1	100%	1	2			4	100%	4	1	33%	2	67%	3	6	55%	11	100%	11
Youtube		-		-					1	100%	1	2	100%			2	13	59%	16	73%	22
Zedge	1	100%			1				1	100%	1			3	100%	3	4	40%	9	90%	10
Total	**8**	**42%**	**14**	**74%**	**19**	**15**	**5**	**25%**	**18**	**90%**	**20**	**13**	**43%**	**20**	**67%**	**30**	**92**	**61%**	**125**	**82%**	**152**

Table 3.6: Distribution of exception types for all tool versions. Note that each cell shows the number of distinct crashes of the specific type triggered by the corresponding tool on all apps. A blank cell indicates no crash. If an exception type appears only once across all tools and apps, it is counted in "Other" instead of being shown in a separate row. Thus, the numbers of distinct crashes in "Other" also indicate the numbers of exception types. The "Total" column shows the numbers of *distinct* crashes for each exception type across all tools.

Exception Type	Ape_S	Ape_O	Mk	Ch_O	Ch_E	Wt_O	Wt_E	St_O	St_E	Total
`ActivityNotFoundException`	2	5	4	2	4	4	3		3	**11**
`ExceptionInInitializerError`								2	2	**3**
`IllegalArgumentException`					1				1	**2**
`IllegalStateException`	2	3	4		2		4		2	**17**
`NoClassDefFoundError`								2	2	**2**
`NullPointerException`	1	4	2	1	5		9	5	11	**25**
`OutOfMemoryError`			1			2				**3**
`RuntimeException`	2	2	4	2	3	6	2	81	102	**129**
Other	1				3	1	2	2	2	**11**
Total	**8**	**14**	**15**	**5**	**18**	**13**	**20**	**92**	**125**	**203**

can be substantial enough to change the relative competitiveness among tools. For example, Table 3.4 shows that for Stoat, compared with Monkey, the Toller-enhanced version achieves higher average code coverage, while the original version achieves lower average code coverage. In Ape's case, the slow version has much smaller advantages over other tools: its average code coverage

is only about 2% relatively higher than the Toller-enhanced WCTester. We also find that the original Ape achieves higher code coverage than other tools on 10 apps, while the slow Ape does that on only 4 apps. For comparison, the Toller-enhanced Chimp and WCTester top the ranks on 4 and 3 apps, respectively, when we omit the original Ape from consideration. Figure 3.4 additionally shows that the slow Ape (denoted as Ape_S) constantly has comparable average code coverage as the Toller-enhanced WCTester (denoted as WCTester_E), where Ape_S starts to beat WCTester_E after 40 minutes of testing. Interestingly, according to Figure 3.4, the Toller-enhanced WCTester even has higher code coverage than all other tools in the first several minutes of testing. Specifically, when we observe the area under the curve (AUC) for every minute, we find that if developers are given at most 9 minutes to use Android UI test generation tools, then the Toller-enhanced WCTester gives the highest AUC instead of the original Ape.

Analysis of Negative Code Coverage Improvements To better understand our results for this RQ, we manually study some of our results to understand why some tools have negative code coverage improvements on some apps after Toller's infrastructure enhancements. Specifically, we manually look into all of the cases where the coverage decrement is over 1% given that smaller changes (<1%) are likely caused by random noise. We look at tool logs and differences in method coverage to speculate root causes.

We are able to identify only one major cause for the reduced effectiveness: *Unsupported UI element types.* The current implementation of Toller does not support obtaining the inner contents of certain types of UI elements, such as `WebView`s that maintain their own non-standard, internal UI-related data structures. These `WebView`s are a major cause for why apps such as "Filters For Selfie" and "Mirror" have negative code coverage improvements for WCTester. Both of these two apps have Google's AdMob SDK embedded and the SDK relies on `WebView`s to display ads. Without knowing the UI hierarchy inside, it is difficult for tools to produce meaningful UI events to fully exercise this ads-related logic. Future work should explore how Toller can better handle certain types of UI elements (e.g., falling back to UIAutomator for `WebView`s).

3.5.4 RQ3: Crash Triggering Benefits

Table 3.5 shows the cumulative number of distinct crashes (from three runs) for all tool versions evaluated on each app. As shown in the table, the Toller-enhanced versions are capable of substantially improving the total number of distinct crashes, from 5, 13, and 92 to 18, 20, and 125 for Chimp, WCTester, and Stoat, respectively. In Ape's case, the total crash count rises from 8 to 14 by using Ape's improved infrastructure support. Overall, we find that there are 43 pairs of tools and apps with at least one crash (non-empty cells under ΣT). Of the 43 pairs, 30 and 10 pairs have more and fewer (respectively) crashes triggered by enhanced tool versions than original/slow tool versions. The remaining 3 pairs have the same number of crashes for both tool versions. Of the 30

Table 3.7: Cumulative method coverage for all tool versions. Note that Ch, Wt, and St denote Chimp, WCTester, and Stoat, respectively. For each tool T, T_O refers to its original version, while T_E refers to our Toller-enhanced version. Each integer cell under T_O or T_E shows the cumulative number of covered distinct methods across three runs by the corresponding tool version on the corresponding app. Each integer cell under ΣT indicates the union number of covered methods by two tool versions on the corresponding app. $\%_O = T_O/\Sigma T \times 100\%$, similar for $\%_E$. In the 'Average' row, each integer indicates the average value of all numbers in the respective column, while each percentage is calculated from average values using the same methodology as aforementioned.

App Name	Ape_S	$\%_S$	Ape_O	$\%_O$	ΣApe	Ch_O	$\%_O$	Ch_E	$\%_E$	ΣCh	Wt_O	$\%_O$	Wt_E	$\%_E$	ΣWt	St_O	$\%_O$	St_E	$\%_E$	ΣSt
Abs	8872	95%	9318	100%	9348	9077	97%	7479	80%	9369	9107	89%	8205	80%	10206	6367	77%	7380	89%	8290
Duolingo	14789	96%	15018	97%	15471	14282	97%	14235	96%	14770	12086	81%	14136	94%	14989	12397	80%	14937	96%	15558
Filters For Selfie	2380	42%	5684	99%	5730	2343	97%	2286	95%	2407	2759	68%	2177	54%	4067	2479	73%	3365	99%	3406
GoodRx	16451	96%	15504	90%	17191	14973	89%	16214	97%	16762	14383	85%	16615	98%	16911	12257	81%	14173	93%	15185
Google Translate	9709	93%	10292	99%	10428	8545	79%	10537	97%	10829	9732	95%	9689	95%	10192	6519	72%	7665	85%	9000
Marvel Comics	5614	81%	6688	96%	6946	5016	98%	4940	96%	5140	4736	94%	4614	92%	5031	4667	83%	5344	95%	5647
Merriam-Webster	8776	97%	8614	95%	9046	8293	90%	8858	96%	9223	6859	75%	8710	95%	9192	8353	86%	9314	95%	9766
Mirror	1196	93%	1253	97%	1290	1514	99%	796	52%	1534	1309	97%	959	71%	1354	1058	79%	1213	91%	1335
My Baby Piano	1572	31%	5059	100%	5065	1590	87%	1810	100%	1818	2682	66%	1555	38%	4083	1654	100%	253	15%	1662
Sketch	8737	92%	8919	94%	9519	8687	93%	8955	96%	9311	8120	91%	8264	93%	8913	6874	71%	9351	97%	9614
trivago	19999	97%	20437	100%	20524	19857	98%	19980	98%	20342	20010	98%	19847	97%	20399	6072	29%	20926	100%	20981
WEBTOON	23982	85%	27149	97%	28088	18048	63%	27933	98%	28643	19754	68%	27185	93%	29238	9957	36%	25123	91%	27457
Word	13550	93%	13034	90%	14514	11711	78%	14645	97%	15095	13946	94%	13706	93%	14763	11055	80%	13493	98%	13768
Youtube	31010	87%	28268	79%	35681	21122	86%	21427	87%	24572	20101	77%	23709	91%	26069	20866	71%	22451	76%	29427
Zedge	39932	77%	50763	98%	51562	52210	93%	41536	74%	56212	39612	95%	38550	92%	41765	28960	68%	36159	85%	42503
Average	**13771**	**86%**	**15067**	**94%**	**16027**	**13151**	**87%**	**13442**	**89%**	**15068**	**12346**	**85%**	**13195**	**91%**	**14478**	**9302**	**65%**	**12743**	**89%**	**14240**

pairs where the enhanced tool versions have more crashes, 21 pairs' cumulative crashes are all from the enhanced tool versions (highlighted cells under T_E and Ape_O). On the other hand, of the 10 pairs where the enhanced tool versions have fewer crashes, only 6 pairs' cumulative crashes are all from the original/slow versions. In general, when the original/slow versions trigger more crashes, the differences are generally small: only one crash for 7 of the 10 pairs. Randomness in the tools' and apps' logic is likely responsible for why original/slow versions can trigger more distinct crashes than Toller-enhanced versions. Overall, our results find that infrastructure enhancements help test generation tools with not only covering more code but also triggering more distinct crashes.

Exception Types We additionally study the distribution of exception types. As shown in Table 3.6, the Toller-enhanced versions trigger not only more instances of crashes, but also more types of exceptions (the number of non-empty cells): from 3, 4, and 6 types to 8, 6, and 9 types on Chimp, WCTester, and Stoat, respectively. In Ape's case, the slow version triggers 5 types of exceptions, while the original version triggers only 4 types. One possible explanation for this finding is the randomness in the tools' and apps' logic. Nevertheless, our results still find that infrastructure enhancements help most test generation tools trigger more distinct types of crashes.

3.5.5 RQ4: Overlap of Code Coverage and Crashes

In RQ4, we investigate whether the code coverage achieved and crashes triggered by tools with infrastructure enhancements are subsumed by what the original/slow versions of the tools achieve

and trigger. To answer this RQ, we measure the overlaps between code coverage achieved and distinct crashes triggered by both versions of each tool.

Table 3.7 shows the cumulative code coverage for all tool versions. Specifically, for each tool on each app, this table shows how many methods are covered by either version in any run, as well as how many methods can be covered by only one of the versions. As shown in the table, the Toller-enhanced versions achieve, on average, 89%, 91%, and 89% coverage of all methods that can be covered by either version of Chimp, WCTester, and Stoat, respectively. In Ape's case, the original version achieves 94% coverage of all methods that can be covered by either the slow or original version. Our results suggest that tool versions with infrastructure enhancements can generally replace the original versions as the versions with enhancements provide the most of the coverage achievable by either version.

We use the same methodology to show the overlaps of crashes triggered by the two versions of each tool. As shown in Table 3.5, the Toller-enhanced versions also cover most of the crashes triggered by either version shown by the $\%_E$ columns in the table. In fact, 90%, 67%, and 82% of the cumulative distinct crashes detected by Chimp, WCTester, and Stoat, respectively, are triggered by the Toller versions. For Ape, the original version covers 74% of all crashes. Our results again suggest that tool versions with infrastructure enhancements can generally replace the original versions as the enhanced versions provide the most of the detected crashes.

3.5.6 RQ5: Effectiveness of Enhancements

To understand how Toller's two enhancements have contributed to test generation tools' code coverage and crash triggering ability, we additionally conduct experiments by enabling only UI Hierarchy Capturing and comparing its results with the setting where both enhancements are used (results in Sections 3.5.3 and 3.5.4). We do not evaluate only UI Event Execution, as Toller's UI Event Execution implementation depends on UI Hierarchy Capturing and does not work on its own as discussed in Section 3.4. Tables 3.8 and 3.9 show the average method coverage and the number of distinct crashes, respectively, for all test generation tools using different enhancement options under the same experimental settings. More detailed experiment data is available on our website [52].

As shown in Tables 3.8 and 3.9, enhancing UI Hierarchy Capturing already improves both achieved code coverage and triggered crashes, while enhancing UI Event Execution leads to even better testing effectiveness, particularly for the number of distinct crashes triggered. One interesting finding is that for Chimp and WCTester, the UI Event Execution enhancement does not improve the overall code coverage. Beyond the fact that all tools are likely to cover less new code as time increases, we identify multiple additional causes for why UI Event Execution may not increase code coverage:

Table 3.8: Average method coverage for enhancements. Note that "None" denotes that no infrastructure enhancement is applied. "HC Only" denotes that only UI Hierarchy Capturing enhancement is used, while "HC + EE" denotes that both UI Hierarchy Capturing and UI Event Execution enhancements are used.

	None	HC Only	Δ	HC + EE	Δ
Chimp	10794	12072	11.8%	12071	11.8%
WCTester	10973	12112	10.4%	12116	10.4%
Stoat	6114	10121	65.5%	10399	70.1%

Table 3.9: # of distinct crashes for enhancements

	None	HC Only	Δ	HC + EE	Δ
Chimp	5	8	1.6x	18	3.6x
WCTester	13	15	1.2x	20	1.5x
Stoat	92	103	1.1x	125	1.4x

- As discussed in Section 3.4, UI Event Execution skips the logic of dispatching low-level UI events on UI elements, likely resulting in the loss of coverage. We mitigate this limitation by falling back to low-level UI event injection when we find an event handler that has not been exercised. However, the strategy could still miss edge cases (e.g., when different UI elements share the same parameterized event handler class).

- Faster UI Event Execution and faster UI Hierarchy Capturing can result in higher CPU usages and overload the emulators, likely causing apps to stop responding.

- Tools might not be accustomed to both fast UI Event Execution and fast UI Hierarchy Capturing. Being unaccustomed to both may cause the tools to be too fast when an app loads content asynchronously, and the time overhead incurred by slower UI Event Execution or UI Hierarchy Capturing actually helps the tools properly wait for the content to load. Such cases are known to cause UI flaky tests [56].

Future work should explore how to carefully design solutions to address the aforementioned causes. For example, future work can intelligently decide on the waiting time at each step to mitigate the effects of device overloading or asynchronous loading with minimal unnecessary waiting costs.

3.6 THREATS TO VALIDITY

The internal threats to the validity of our work are that Toller's implementation and the scripts used to generate the tables and figures could have faults that might have affected our

results. Furthermore, our setup of the Android test generation tools used in our experiments could have been incorrect and affected our results. To mitigate these internal threats to validity, we design our experiments to output extensive logs along with the metrics used in our experiments. We then manually analyze a sample of the logs from our experiments to ensure that the presented results match what we observe from the logs.

The main external threat to the validity of our work is the representativeness of the apps and the Android UI test generation tools selected for our experiments. To mitigate this threat to validity, we select the top apps from 13 different categories of apps on the Google Play Store. Therefore, the selected apps vary greatly in their functionality and APK size (from 3.3MB to 93MB). The Android test generation tools used for our experiments are from a previous study [20] of Android UI test generation tools. The study finds that Monkey is the best Android UI test generation tool among six tools. Among these tools, we find that two are runnable on our infrastructure and do not require app instrumentation. To demonstrate the improvements that Toller can have on Android UI test generation tools, particularly on ones that use UIAutomator, we select the two tools from the previous study and implement a version of Monkey, known as Chimp, that uses UIAutomator for our experiments.

Another threat to the validity of our work is the randomness from the Android UI test generation tools, apps, and emulators. Namely, across different runs of the same tool, app, and emulator, the obtained metrics could change. To mitigate this threat to validity, we run each pair of tools and apps three times, where each run is performed on a newly-created emulator with the same software and hardware configurations throughout all of the experiments. The conclusions that we make from our results are then from the aggregation of the three runs for each pair of tools and apps.

3.7 DISCUSSION

Modifying Android OS. For our experiments, we modify AOSP Android 6.0 on both emulators and real devices. Our modified emulator image is publicly available [52] as a portable testing environment for others to immediately begin using.

While modifying the Android framework eliminates the risks of app instrumentation, it is true that modifying the Android framework can also be undesirable. For instance, we might fail to modify a customized Android OS. Additionally, the modification usually requires root access to the testing device, not being always feasible. To support developers who may be interested in infrastructure enhancements without modifying the Android framework, we also design Toller so that it can be bundled with the target app's code through source code integration or binary instrumentation. Because Toller relies on only Android framework classes, it is only necessary to inject a startup method call into existing app code to make Toller work.

On the other hand, we argue that different testing needs should be satisfied with different ways

of support. Specifically, the Android-framework-based solution is suitable for external testing on certain devices, such as app examinations conducted by app marketplaces. The code-bundling-based solution may be more suitable for in-house testing conducted by app developers, such as compatibility testing that involves various devices.

3.8 SUMMARY

Much work has been proposed by researchers to improve Android UI test generation tools with sophisticated algorithmic designs. Recent studies have shown that these tools barely outperform (w.r.t. code coverage and crash triggering ability) Monkey, a simple tool that generates and injects purely randomized UI events. To understand the inefficiencies of Android test generation tools, we have conducted a motivating study to determine the sources and extents of the inefficiencies for these tools. Our motivating study has found that capturing information about the contents on the screen (*UI Hierarchy Capturing*) and executing UI events (*UI Event Execution*, such as clicks) use on average 70% of the testing run time. Based on our findings, we have proposed Toller, a tool to provide efficient infrastructure support for UI Hierarchy Capturing and UI Event Execution to Android UI test generation tools. Our experiments show that Toller can substantially (1) reduce the run time used by the infrastructure that the test generation tools depend on and (2) improve the code coverage and crash triggering ability of these tools when they are given a reasonable amount of run time. We make the source code of Toller and the scripts used to set up Toller publicly available [52]. We hope that our results can raise the community's awareness of the significance of infrastructure support beyond the community's existing heavy focus on algorithms.

CHAPTER 4: VET: PROVIDING EXPLORATION GUIDANCE VIA IDENTIFYING AND AVOIDING UI EXPLORATION TARPITS

4.1 OVERVIEW

In this chapter, we mainly focus on detecting and taming the factors that lead to the ineffectiveness of existing mobile UI test generation tools. We find that existing mobile UI test generation tools are often prone to *exploration tarpits*[1], where tools get stuck with a small fraction of app functionalities for an extensive amount of time. We show a real-world example in Section 4.2, where a state-of-the-art Android UI test generation tool named Ape [15] decides to log itself out one minute after testing an app starts, without being able to log back in, and since then gets stuck with exploring the app's pre-login functionalities (i.e., exploration tarpits) instead of its main functionalities. It is possible that tool vendors/users manually hardcode rules for the tools to avoid specific exploration tarpits, such as instructing Ape to avoid tapping the "logout" button or writing a script to support automatic login. However, these rules can hardly generalize, being fragile in face of diverted testing environments (e.g., unreliable network to process login requests), fast app iterations, and the demand of batch testing product lines. Our findings in Section 4.5.2 show various cases as such where exploration tarpits can be caused by unexpected flaws in a tool's exploration strategies or implementation defects.

To automatically identify and resolve exploration tarpits, in this chapter, we propose a general approach and its supporting system named VET for the given specific Android UI test generation tool on the given specific app under test (AUT). VET works in three stages. (1) VET runs the tool on the AUT for some time and records the interactions between the tool and AUT, in the form of UI *traces*. A UI trace consists of app UIs interleaving with the actions taken by the tool. (2) VET then analyzes the collected traces to identify trace subsequences (termed *regions*) that manifest exploration tarpits. (3) VET guides the tool in subsequent test runs to prevent or recover from an exploration tarpit by monitoring the testing progress and taking actions based on findings from the identified regions.

VET includes two specialized algorithms targeting two corresponding patterns of exploration tarpits: *Exploration Space Partition* and *Excessive Local Exploration* (see Section 4.2 and Section 4.4). Exploration Space Partition, corresponding to Figure 4.1a, indicates that the fraction of app functionalities explored by the tool is disconnected from most of the app functionalities after some specific action (e.g., tapping "OK" in Screen C). Such situations can be prevented by disabling the aforementioned action. Excessive Local Exploration indicates that the tool enters a hard-to-escape fraction of the app UIs and needs a significant amount of time to reach other functionalities, as demonstrated in Figure 4.1b. This issue can be addressed by either preventing the tool from

[1]The name of exploration tarpits is inspired by the Mythical Man-Month book [57].

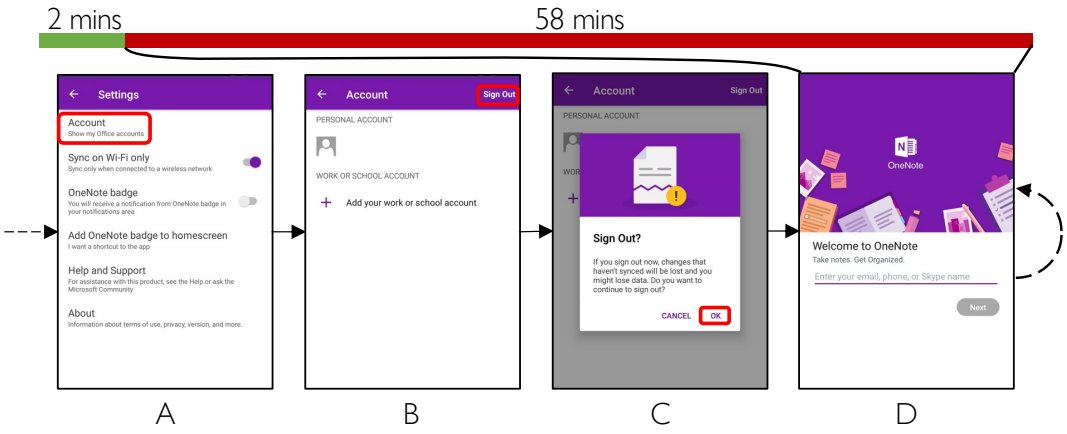
entering (e.g., disabling "START" in Screen E in Figure 4.1b) or assisting the tool to escape (e.g., restart the app upon observation of Screen F). To design the two algorithms, we first construct fitness value formulas that quantify how well a region on the given trace matches a targeted pattern. We then apply fitness value optimization on the entire trace to determine the region that best fits our targeted patterns.

We evaluate VET using three state-of-the-art Android UI test generation tools (Monkey [7], Ape [15], and WCTester [32, 33]) with 16 widely used industrial apps. We collect 144 traces by running each tool on each app three times for one hour each (*original* runs). VET reports at least one exploration tarpit region in each (tool, app) pair, with 131 regions in total, each spanning about 27 minutes on average. The longest regions span over 59 minutes, about 98.6% of the one-hour testing time budget. After inspecting the 131 reported regions, we confirm the root causes of 96 regions, including both limitations of UI exploration strategies (e.g., early logouts) and defects in tool implementation (e.g., hanging), as shown in Section 4.5.2. We then perform six other one-hour runs for each (tool, app) pair: (1) three *guided* runs using VET to automatically avoid all the exploration tarpit regions identified in the original runs during testing on three runs, and (2) three *comparison* runs not using VET.

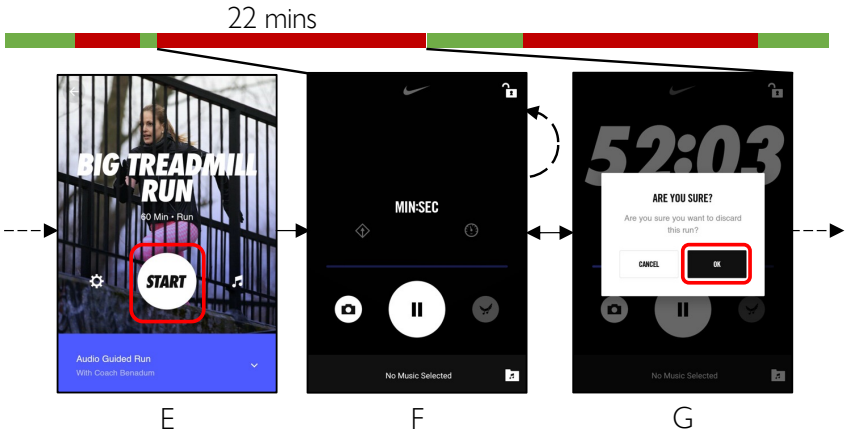
Based on the preceding evaluation setup, we compare the code coverage (of the given app) achieved by applying each tool with and without the assistance of VET given the same time budget. Specifically, we compare the combined code coverage and the numbers of distinct crashes for (1) original runs and guided runs, and (2) original runs and comparison runs. The evaluation results show that on average a tool assisted by VET achieves up to a 15.3% relative code coverage increment and triggers up to 2.1x distinct crashes than the tool without the assistance of VET.

In summary, this chapter makes the following main contributions:

- A new perspective of improving the given automated UI test generation tool by automatically identifying and addressing exploration tarpits for the given target AUT;
- Algorithms for effective identification of two manifestation patterns of exploration tarpits;
- A practical system [58] that can be automatically applied to enhance any Android UI test generation tool such as Monkey [7], Ape [15], and WCTester [32, 33], on any AUT;
- Comprehensive evaluation of VET, demonstrating that VET reveals various issues related to tools or app usability, and that VET automatically resolves those issues, helping the tools achieve up to a 15.3% relative code coverage increment and 2.1x distinct crashes on 16 popular industrial apps.



(a) Exploration Space Partition



(b) Excessive Local Exploration

Figure 4.1: Motivating examples of exploration tarpits described in Section 4.2. Note that colored bars on the top represent the progress of two 1-hour tests, where green bars refer to normal exploration and red bars refer to exploration tarpits. Dashed straight arrows indicate visiting the screen from some other screen, and solid arrows show transitions between two screens after clicking the red-boxed UI elements. The dashed curve arrow on Screen D depicts that Ape cycles around D until the end of testing. The dashed curve arrow on Screen F shows that Monkey stays on F within the 22-minute exploration tarpit window.

4.2 MOTIVATING EXAMPLES

We present two concrete examples from our experiments covering Exploration Space Partition and Excessive Local Exploration (see Section 4.4). These examples provide contexts for further discussion and help illustrate the motivations that drive the design of VET.

4.2.1 Exploration Space Partition

We run Ape [15], a state-of-the-art Android UI test generation tool to test a popular app, Microsoft *OneNote*. The result is illustrated in Figure 4.1a. We manually set up the account to log in to the app's main functionalities, and then start Ape. We run Ape without interruptions for one hour and check the test results afterward.

In the one-hour testing period, Ape explores only 12% (9 out of 76) activities. To understand the low testing effectiveness, we investigate the UI trace captured during testing and find the root cause to be exploration tarpits:

1. Ape performs exploration around OneNote's main functionalities for about two minutes, covering 7 (out of 9) of all the activities covered in the entire one-hour test run. We omit this phase in Figure 4.1a.

2. About two minutes after testing starts, Ape arrives at the "Settings" screen (Screen A) and decides to click "Account" (the red-boxed UI element) for further exploration.

3. Ape arrives at the "Account" screen (Screen B) and clicks the "Sign Out" button. The click pops up a window (Screen C) asking for confirmation of getting logged out.

4. Ape clicks "CANCEL" first, and then goes back to the "Account" screen. However, Ape clicks the "Sign Out" button again, knowing that there is one action not triggered yet in the confirmation dialog. Subsequently, Ape clicks the "OK" button (Screen C) and logs itself out.

5. The logout leads to the entry screen (Screen D). From this point, Ape has access to only a small number of functionalities (e.g., logging in). Ape cannot log in due to the difficulty of auto-generating the username/password of the test account. In the remaining 58 minutes, Ape explores two new activities in total.

This example represents Exploration Space Partition described in Section 4.1. The essential problem is that Ape does not understand UI semantics—it does not know that the majority of OneNote's functionalities will be unreachable by clicking the "OK" button at the time of action.

4.2.2 Excessive Local Exploration

Figure 4.1b presents another example in which we run Monkey [7], a widely adopted tool, to test another popular industry-quality app, *Nike Run Club*. In this example, Monkey spends about 22 minutes trying to saturate one of the app's functionalities. After investigating into the collected UI trace, we find the following behavior when Monkey interacts with the app:

1. Monkey explores other functionalities normally before entering Screen E that allows the tool to enter the functionality where the tool later gets trapped. We name the functionality *the trapping*

functionality. Monkey clicks the "START" button and enters the trapping functionality (Screen F).

2. Monkey keeps clicking around in the trapping functionality. To escape from the trapping functionality, Monkey first needs to press the Back button, and a confirmation dialog (Screen G) will pop up. Monkey then has to click the "OK" button to finish escaping. However, due to being widget-oblivion, Monkey clicks only randomly on the screen, resulting in constant failures to click "OK" when the confirmation dialog is shown. Furthermore, the dialog disappears when Monkey clicks outside of its boundary, and Monkey needs to press the Back button again to make the confirmation show up one more time. It takes 22 minutes for Monkey to find and execute an effective escaping UI event sequence and finally leave the trapping functionality.

3. The aforementioned behaviors are repeatedly observed in the trace (with different amounts of time used for escaping).

This example represents Excessive Local Exploration behavior described in Section 4.1. The essential problem is that Monkey is both widget- and state-oblivion, i.e., the tool is unable to locate actionable UI elements efficiently or sense whether it has been trapped and react accordingly (e.g., by restarting the target app).

4.2.3 Implications

To prevent such undesirable exploration behaviors, a conceptually simple idea is to de-prioritize exploring the entries to aforementioned trapping states (i.e., the "OK" button in Screen C, and "START" button in Screen F). One potential solution is to develop natural language processing (NLP) or image processing based approaches that can infer the semantics of UI elements [59, 60, 61, 62]. While solutions based on understanding UI semantics are revolutionary, they are challenging due to fundamental difficulties rooting in NLP and image processing.

In this chapter, we explore a more practical and evolutionary solution based on understanding exploration tarpits by mining UI traces. We show that it is feasible to identify the existence and location of such behavior through pattern analysis on interaction history. Given the location of exploration tarpits, we can further identify which UI actions might have led to such behavior. Taking the example of Figure 4.1, Ape starts to visit a very different set of screens (e.g., the welcome screens in Screen D in Figure 4.1a) after clicking "OK", and the number of explored screens dramatically decreases. Therefore, we can look at the screen history and find the time point where the symptom starts to appear. The UI action located at the aforementioned time point is then likely the cause of the symptom. Our VET system uses a specialized algorithm (Section 4.4.2) to effectively locate the starting time point of exploration tarpits similar to the aforementioned instance.

4.3 BACKGROUND

This section presents background knowledge about UI hierarchy to help readers understand our algorithm design and implementations in the scope of Android UI testing.

A *UI hierarchy* structurally represents the contents of app UI shown at a time. Each UI hierarchy consists of *UI properties* (e.g., location, size) for individual *UI elements* (e.g., buttons, textboxes) and hierarchical relations among UI elements. On Android, each activity internally maintains the data structure for its current UI hierarchy. Typically, UI elements are represented by `View` [63] subclass instances, and hierarchical relations are represented by child `Views` of `ViewGroup` [64] subclass instances.

A key component of UI testing is to identify the current app functionality. The functionality is identified by *equivalence check for UI hierarchies*, because UI hierarchies are usually used as indicators of apps' functionality scenarios. Thus, checking the equivalence between the current and past UI hierarchies allows tools to identify whether a new functionality is being exercised. If the current functionality has been covered, the tool can additionally leverage the knowledge associated with the functionality to decide on the next actions. There are different ways to check UI hierarchy equivalence:

- **Strict comparison.** A simple way to check the equivalence of two UI hierarchies is to compare their UI element trees and see whether they have identical structure and UI properties at each node. In practice, such simple equivalence checking is too strict. For example, on an app accepting text inputs, a tool checking exact equivalence can count a new functionality every time one character is typed.

- **Checking similarity.** A workaround to the aforementioned issue of strict comparison is to check similarities of two UI hierarchies against a threshold. However, ambiguity can become the new issue, given that the similarity relation is not transitive: suppose that A is similar to both B and C, it is still possible that B is not similar to C. Then if both B and C are in the history (regarded as different functionalities), and A comes as a new UI hierarchy, the tool is unable to decide on which functionality to use the associated knowledge from. To fix the ambiguity issue, we can perform *screen clustering*, essentially putting mutually similar screens into individual groups and regarding each group as representing one single functionality. Then the downside is that screen clustering can be a computationally expensive operation, especially for traces with many screens.

- **Comparing abstractions.** A more advanced solution is to check the equivalence at an abstraction level, employed by many model-based UI test generation tools [11, 13, 15, 65, 66]. In the previous example, one can leave out all user-controlled textual UI properties from the hierarchy and the equivalence check can tell that the tool is staying on the same screen regardless of what has been entered. While abstracting UI hierarchies is conceptually effective, it is challenging

to design effective UI abstraction functions. The difficulty lies in identifying UI properties or structural information to distinguish different app functionalities, especially when screens have variants with relatively subtle differences.

Ape [15] includes adaptive abstractions to address the challenge of automatically finding proper UI abstraction functions in different scenarios. Ape dynamically adjusts its abstraction strategy (e.g., which UI property values should be preserved) during testing based on feedback from strategy execution (e.g., whether invoking actions on UI hierarchies with the same abstraction yields the same results). Unfortunately, the adaptive abstraction idea assumes the availability of sufficiently diverse execution history for feedback, and such history is not always available when analyzing given traces as in our situation.

Given the pros and cons of the aforementioned ways, we empirically adopt a hybrid approach for UI hierarchy equivalence check. First, we always abstract UI hierarchies: (1) we consider only visible UI elements (i.e., `View.getVisibility() == VISIBLE`, and the element's bounding box intersects with its parent's screen region), (2) we keep the activity ID and the original hierarchical relations among UI elements, and (3) we retain only UI element types and IDs from UI properties. Second, we check the similarities of abstract UI hierarchies and cluster them into groups only when the analysis is sensitive to the absolute number of distinct screens. More details on achieving clustering efficiently are elaborated in Section 4.4.3.

4.4 THE VET APPROACH

4.4.1 Overview

We propose VET, a general approach and its supporting system that automatically identifies and addresses exploration tarpits for *any* given Android UI test generation tool on *any* given AUT. Our implementation of VET is publicly available at [58].

As illustrated in Figure 4.2, for a given tool and AUT, VET works in three stages. First, VET runs the target tool on the AUT for a certain amount of time and records the interactions between the tool and AUT. With help from our Android framework extension TOLLER [22], VET collects *trace*(s) that consist of AUT UIs interleaving with the tool's actions. Then, VET analyzes each individual trace with specialized algorithms to identify trace subsequences (termed *regions*) that manifest the tool's exploration tarpits. Optionally, one can rank the identified regions based on their time lengths, where longer regions receive higher ranks, to prioritize regions that are likely to exhibit exploration tarpits with higher impacts (see Section 4.5.2). Finally, VET learns from the identified regions and guides the tool in subsequent runs to avoid exploration tarpits, by monitoring the testing progress and taking actions based on findings from the identified regions. With the support from TOLLER, VET is currently capable of (1) preventing specified actions by

disabling the corresponding UI elements at runtime and (2) assisting the AUT to escape from the specified screens by restarting the AUT. The identified regions additionally support manual investigations of testing efficacy by providing localization help.

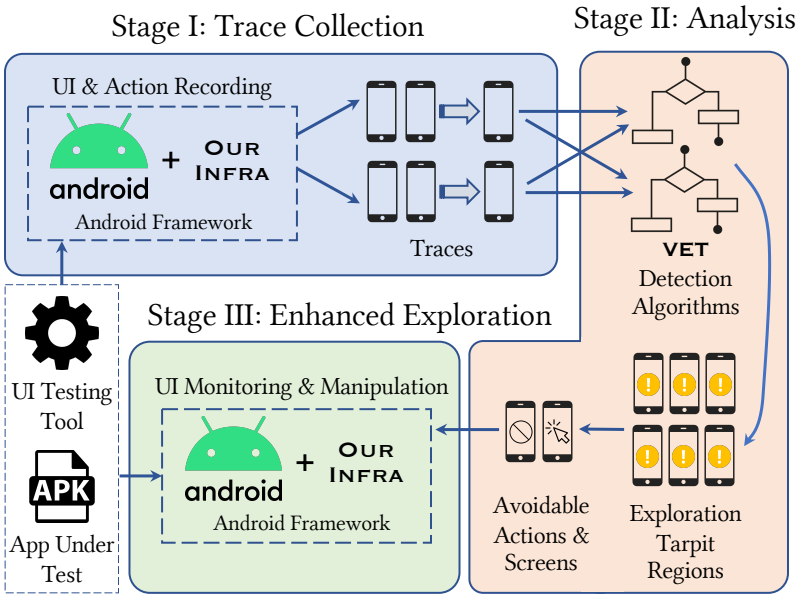


Figure 4.2: Overview of Vet.

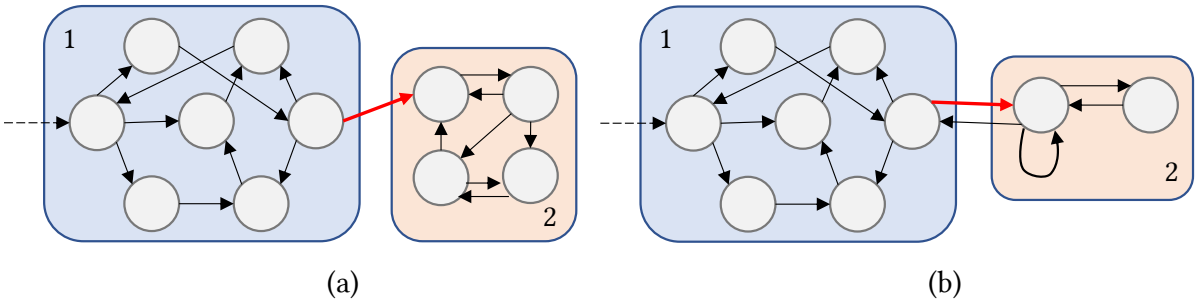


Figure 4.3: Two patterns of exploration tarpits. Note that each subgraph corresponds to an example trace, where each circle represents a distinct screen in the trace (e.g., each subfigure in Figure 4.1), each arrow indicates that action(s) is observed between two screens in the trace, and each curved rectangle depicts a UI subspace. Red arrows denote destructive actions (e.g., clicking "OK" in Screen C of Figure 4.1a, "START" in Screen E of Figure 4.1b) while dashed arrows show where traces begin.

We equip Vet with two specialized algorithms targeting two patterns of exploration tarpits: *Exploration Space Partition* and *Excessive Local Exploration*. Characteristics of the two algorithms' targeted patterns are illustrated in Figure 4.3 and discussed as follows:

- *Exploration Space Partition.* As shown in Figure 4.3a, the UI test generation tool traverses through a UI subspace (Subspace 2) for a long time after the execution of some action (the

red arrow), and the tool is *unable* to return to the previously visited UI subspace (Subspace 1). Furthermore, the tool visits much fewer distinct screens after the action. The presence of the symptom suggests that the tool has triggered a destructive action (effectively the partition boundary of the entire trace and beginning of the exploration tarpit) that prevents the tool from further exploring the app's major functionalities. The first motivating example from Section 4.2 corresponds to this symptom, where clicking the "OK" button is the destructive action that gets Ape trapped in multiple screens related to logging in (Subspace 2) and prevents Ape from further accessing OneNote's main functionalities (Subspace 1).

- *Excessive Local Exploration*. As shown in Figure 4.3b, the UI test generation tool is trapped in a small UI subspace (Subspace 2) for an extended amount of time after the execution of the corresponding destructive action (the red arrow). However, the tool is *capable* of returning to the previously visited UI subspace (Subspace 1) despite the difficulties. It is also likely that the tool will get trapped again within Subspace 2 after returning to Subspace 1. Consequently, the tool spends an excessive amount of time repetitively testing limited functionalities in this hard-to-escape subspace. The second motivating example from Section 4.2 corresponds to this symptom, where clicking the "START" button gets Monkey trapped in Screens F and G (Subspace 2). Clicking "OK" helps Monkey go back to Screen E (within Subspace 1) and other functionalities, but it does not take a long time before the tool gets trapped again within Subspace 2.

As can be seen, Exploration Space Partition targets higher-level irreversible transition of UI exploration space, while Excessive Local Exploration focuses on lower-level difficulties of exercising a specific functionality. Note that it is possible for the regions reported by the two algorithms on the same trace to overlap. For example, Excessive Local Exploration might also capture exploration tarpits within Exploration Space Partition's trapped UI subspace (corresponding to Subspace 2 in Figure 4.3). Such overlaps do not prevent us from finding meaningful targeted exploration tarpits: different exploration tarpits revealed by regions identified by both algorithms suggest the existence of different exploration difficulties.

In the remaining of this section, we describe the two algorithms for capturing Exploration Space Partition and Excessive Local Exploration in Section 4.4.2 and Section 4.4.3, respectively. We show that pattern capturing can be expressed as optimization problems. Table 4.1 describes the notations used to describe VET's algorithms.

4.4.2 Capturing Exploration Space Partition

According to our introductions of Exploration Space Partition, we need to find a destructive action exerted on screen S_n as the partition boundary such that the aforementioned characteristics from Section 4.4.1 can be best reflected. For instance, considering our first motivating example in

Table 4.1: Notations and descriptions used in the algorithms

Notation	*Description*
S_i	Screen $\#i$ in the trace represented by the UI hierarchy.
t_i	The timestamp of screen S_i being observed.
$S_{l,r}$	A region of screens starting at Screen $\#l$ and ending at Screen $\#r$ (with both ends included).
$\{S_{l,r}\}$	The set of distinct screens from $S_{l,r}$ by de-duplicating their UI hierarchies.
$\lvert S^s_{l,r}\rvert$	The number of occurrences of s in $S_{l,r}$.
$t_{\min}$	A predefined threshold that decides the minimum time length of any $S_{l,r}$ (i.e., $t_r - t_l \geqslant t_{\min}$) that may be included in algorithm outputs.

Section 4.2, we hope to pick up the screen shown in Figure 4.1c as S_n. We optimize the following formula to find the most desirable S_n from a trace with N screens:

$$\underset{1 \leqslant n < E_p}{\arg\min} \quad [\sum_{s \in \{S_{1,n}\}} \frac{|S^s_{n+1,N}|}{N-n}] + 2 \cdot \sigma(\frac{|\{S_{n+1,N}\}|}{|\{S_{E_p+1,N}\}|} - 1) - 1 \tag{4.1}$$

In the formula, E_p is a pre-calculated limit indicating the upper bound of n during optimization, and σ denotes the Sigmoid function. Note that $N - n$ can be pulled out of the sum subformula. The intuition of the formula design is as follows:

1. As part of the characteristics, the tool should ideally be able to visit few to no screens that have appeared no later than S_n after the tool passes S_n. Correspondingly, in our motivating example, screens shown before Figure 4.1c (depicting the app's main functionalities) are dramatically different from the screens afterward (logging in, ToS, etc.). In the formula, the nominator of the first term (intended to be minimized) quantifies the proportion of screens seen before S_n within $S_{n+1,N}$.

2. As the denominator of the first term, $N - n$ essentially calculates how many (non-distinct) screens the tool visits after S_n. There are two purposes of this design. First, we hope to normalize the first term in the formula (so that two terms can weigh the same). Given that $\sum_{s \in \{S_{1,n}\}} |S^s_{n+1,N}| = \sum_{s \in \{S_{1,n}\} \cap \{S_{n+1,N}\}} |S^s_{n+1,N}| \leqslant \sum_{s \in \{S_{n+1,N}\}} |S^s_{n+1,N}| = N - n$, the first term is guaranteed to fall within $[0, 1]$. Second, we want to push S_n backward (note that smaller n makes the first term smaller) because we assume that the design makes S_n closer to the exploration tarpit's root cause, which should appear earlier than other causes.

3. As another part of the characteristics, the tool stays within a certain UI subspace for a long time; thus, the tool will go through screens within the subspace very often. If the tool generally uniformly visits most or all distinct screens within the subspace, by observing a small period of

exploration (corresponding to $S_{E_p+1,N}$ in the formula) we should have a fairly precise estimation (i.e., $\{S_{E_p+1,N}\}$) of the subspace boundary, which is characterized by $\{S_{n+1,N}\}$. The second term in the formula corresponds to this intuition, where the closer $\{S_{E_p+1,N}\}$ is to $\{S_{n+1,N}\}$ (note that $\{S_{E_p+1,N}\} \subseteq \{S_{n+1,N}\}$), the more favorable it becomes during optimization.

4. By setting an upper bound E_p on n and regularizing the ratio with a Sigmoid function and applying appropriate linear transformations, we can guarantee that the second term in the formula always ranges from 0 to 1, being the same as the first term. In the end, two terms in the formula contribute equally to optimization choices.

To determine E_p on each trace, because our optimization scope does not include any interval shorter than $[E_p, N]$, we choose a value such that $t_N - t_{E_p}$ is closest to $t_{\min}$.

After obtaining a potentially suitable S_n through optimizing the aforementioned formula, we additionally check whether $|\{S_{1,n}\}| > |\{S_{n+1,N}\}|$ is satisfied, essentially enforcing the property that the exploration space should be smaller after the partition. Finally, the reported region is $S_{n+1,N}$.

4.4.3 Capturing Excessive Local Exploration

Based on the characteristics of Excessive Local Exploration from Section 4.4.1, we should track the presence of a region showing that the tool is trapped within a small UI subspace for an extended amount of time. For our second motivating example in Section 4.2, one valid choice is the 22-minute region starting from the button click in Screen E of Figure 4.1b. We accordingly optimize the following formula to find the boundaries S_l and S_r of the most suitable region on a trace:

$$\underset{1 \leqslant l \leqslant r \leqslant N}{\arg\min} \frac{|\{\textsc{Map}(S_{l,r}, \textsc{Merge}(\{S_{l,r}\}))\}|}{r - l + 1} \tag{4.2}$$

In the formula, Merge denotes the operation of merging similar screens (this operation will be introduced later). As the optimization formula suggests, we hope to find a suitable region such that it covers few distinct screen groups despite that the tool tries to explore diligently (by injecting numerous actions quantified by $r - l + 1$). Then if $t_r - t_l \geqslant t_{\min}$, we regard that the exploration tarpit region $S_{l,r}$ can be reported. Accordingly in our motivating example, the choice of S_l is Screen F of Figure 4.1b and S_r is the last instance of Screen G of Figure 4.1b in the 22-minute region. S_{l-1} corresponds to Screen E of Figure 4.1b, and the destructive action is reported.

Note that there can be more than one region exhibiting Excessive Local Exploration behavior within a single trace, given the possibility for the tool to escape the UI subspaces where Excessive Local Exploration behavior is observed. In order to find all potential regions, we iterate the

Algorithm 4.1: MERGE: Merging similar screens into groups (each group is represented by its root screen in R)

Input: A set of abstract UI hierarchies H
Output: A mapping $R : H \mapsto R$, where $R \subseteq H$
Sort $h \in H$ by $|h|$ in ascending order
$R \leftarrow \{\}$
foreach $h \in H$ **do**
  $R[h] \leftarrow$ nil
end
foreach $h \in H$ **do**
  if $R[h] = nil$ **then**
    $R[h] \leftarrow h$
    foreach $h' \in H$ **do**
      if $R[h'] =$ nil $\wedge$ SIMCHECK(h, h') **then**
        $R[h'] \leftarrow h$
      end
    end
  end
end
return R

aforementioned optimization process on remaining region(s) each time after one region is chosen, until no more region can be divided.

Design of Merge. As mentioned in Section 4.3, involving screen merging is especially useful for handling Excessive Local Exploration, given that the aforementioned optimization formula is very sensitive to the absolute numbers of distinct screens. Being part of the challenge, an efficient (and mostly effective) screen merging algorithm requires careful design. Given a set of distinct abstract screens (represented by UI hierarchies) to merge, a relatively straightforward (and precise) approach is to first calculate the tree editing distance [67] for each pair of abstract UI hierarchies for similarity check, and then use combinatorial optimization [68] to decide the optimal grouping strategy (e.g., by converting to an Integer Linear Programming [69] problem), such that all screens within the same group are mutually similar and the total number of groups is minimal. Unfortunately, such an algorithm requires exponential time in regards to the number of distinct abstract screens to merge. The design will likely fall short on traces collected using industrial-quality apps, from which we can easily capture hundreds to thousands of distinct abstract screens.

Aiming to make the algorithm practically efficient, we relax the definition of similarity and the goal of optimization from the aforementioned merging algorithm based on insights from our observations. Specifically, we find that in many cases, similar screens can be seen as screen variants derived from base screens by inserting a small number of leaf nodes or subtrees into the abstract UI hierarchy. Based on this assumption with some tolerance for inaccuracy, we can (1) design a more efficient tree similarity checker (Algorithm 4.2), which considers only node insertion distances and

Algorithm 4.2: SimCheck: UI hierarchy similarity checker

Input: Abstract UI hierarchies h_1, h_2
Output: Whether h_1, h_2 are similar enough
Const: Max allowed distance $d_{\max}$, empirically set to 3
$seq_1 \leftarrow []$
foreach $node \in \text{DepthFirstTraverse}(h_1)$ **do**
  $seq_1 \leftarrow seq_1 :: (\text{Props}(node), \text{Depth}(node))$
  // Props obtains the node's UI properties that are preserved during abstraction. See more details in Section 4.3.
end
$seq_2 \leftarrow []$
foreach $node \in \text{DepthFirstTraverse}(h_2)$ **do**
  $seq_2 \leftarrow seq_2 :: (\text{Props}(node), \text{Depth}(node))$
end
$lcs \leftarrow \text{LongestCommonSequence}(seq_1, seq_2)$
if $|lcs| < \min(|h_1|, |h_2|)$ **then**
  return `false`
else
  return $\max(|h_1|, |h_2|) - |lcs| \leqslant d_{\max}$
end

has $\Theta(|h_1| \cdot |h_2|)$ time complexity (compared with $O(|h_1| \cdot |h_2| \cdot \text{Height}(h_1) \cdot \text{Height}(h_2))$ for full tree edit distance), and (2) replace the inefficient combinatorial optimization with a highly efficient greedy algorithm (Algorithm 4.1), which tries to find all the base screens with $O(|H|^2 \cdot \max_{h \in H} |h|^2)$ time complexity. In practice, with multiple other optimizations not affecting the level of time complexity, the algorithm needs only several seconds on average to process a trace. Even for a very long trace with 2,000 distinct abstract screens and tens of thousands of concrete screens, the algorithm runs for only several minutes.

4.5 EVALUATION

Our evaluation answers the following research questions:

- **RQ1:** How effectively can Vet help reveal Android UI test generation tool issues with the identified exploration tarpit regions?
- **RQ2:** What is the extent of effectiveness improvement of Android UI test generation tools through automatic enhancement by Vet?
- **RQ3:** How likely do Vet algorithms miss tool issues in their identified exploration tarpit regions?

Table 4.2: Overview of industrial apps used for evaluation. Note that '#Install' denotes the approximate number of downloads. 'Login' indicates whether the app requires logging in to access most features.

App Name	Version	Category	#Install	Login
AccuWeather	5.3.5-free	Weather	50m+	✗
AutoScout24	9.3.14	Auto & Vehicles	10m+	✗
Duolingo	3.75.1	Education	100m+	✗
Flipboard	4.1.1	News & Magazines	500m+	✓
Merriam-Webster	4.1.2	Books & Reference	10m+	✗
Nike Run Club	2.14.1	Health & Fitness	10m+	✓
OneNote	16.0.9126	Business	100m+	✓
Quizlet	3.15.2	Education	10m+	✓
Spotify	8.4.48	Music & Audio	100m+	✓
TripAdvisor	25.6.1	Food & Drink	100m+	✓
trivago	4.9.4	Travel & Local	10m+	✗
Wattpad	6.82.0	Books & Reference	100m+	✓
WEBTOON	2.4.3	Comics	10m+	✗
Wish	4.16.5	Shopping	100m+	✓
YouTube	15.35.42	Video Player & Editor	1b+	✗
Zedge	7.2.2	Personalization	100m+	✗

4.5.1 Evaluation Setup

Android UI Testing Tools and Android Apps. We use three state-of-the-art Android UI test generation tools: Monkey [7], Ape [15], and WCTester [32, 33]. We use 16 popular industry Android apps from the Google Play Store, as shown in Table 4.2. These 16 apps are from a previous study [20], which picks the most popular apps from each of the categories on Google Play and compares multiple test generation tools applied on these apps. The apps that we choose need to work properly on our testing infrastructure: (1) they need to provide x86/x64 variants of native libraries (if they have any), (2) they do not constantly crash on our emulators, and (3) Toller is able to obtain UI hierarchies on most of the functionalities. We additionally skip apps that (1) have relatively limited sets of functionalities, or (2) require logging in for access to most features and we are unable to obtain a consistently usable test account (e.g., some apps have disabled our test accounts after some experiments).

Trace Collection. We run each tool on every app for three times to alleviate the potential impacts of non-determinism in testing. Each run takes one hour without interruption, and we restart the tool if it exits before using up the allocated run time. Toller records one UI trace for each test run. While Vet runs separately on each UI trace, results are grouped for each (tool, app) pair. In total, we collect 144 one-hour UI traces from 48 (tool, app) pairs.

Testing Platform. All experiments are conducted on the official Android x64 emulators running Android 6.0 on a server with Xeon E5-2650 v4 processors. Each emulator is allocated

with 4 dedicated CPU cores, 2 GiB of RAM, and 2 GiB of internal storage space. Emulators are stored on a RAM disk and backed by discrete graphics cards for minimal mutual influences caused by disk I/O bottlenecks and CPU-intensive graphical rendering. We manually write auto-login scripts for apps with "Login" ticked in Table 4.2, and each of these scripts is executed only once before the corresponding app starts to be tested in each run. To alleviate the flakiness of these auto-login scripts, we manually check the collected traces afterward and rerun the experiments with failed login attempts.

Overall Statistics. VET reports 131 regions to exhibit exploration tarpits, averaging 2.7 on each (tool, app) pair. Based on VET's reports, the average amount of time involved in exploration tarpits is about 27 minutes per region, with the maximum being 59 minutes and minimum being slightly more than 10 minutes (given that we empirically set $t_{\min} = 10$ minutes for all the experiments).

4.5.2 RQ1. Detected Tool Issues

Methodology We evaluate the effectiveness of VET algorithms in capturing exploration tarpits that reveal issues of test generation tools upon AUTs. Specifically, we first group exploration tarpit regions by the (tool, app) pairs that these regions are observed on. Then, we rank the regions within each (tool, app) pair by their time lengths as mentioned in Section 4.4.1. Finally, we manually investigate each of 131 regions from all (tool, app) pairs. We report any issue for each of these regions with manual judgment. Note that we count only the issue that we consider most specific to the exploration tarpit revealed by each region: if both issues A and B contribute to the exploration tarpit on some region, and A also contributes to other regions on the same trace, we count only B in the statistics.

Results We are able to determine tool issues on 96 of 131 manually investigated regions. Table 4.3 shows the distribution of issue types w.r.t the tool and region ranking. Note that we find each (tool, app) pair to have up to three regions reported by VET; thus, rank-1/2/3 regions cover all 131 regions (with 48/43/40 regions each). The tool issues can be traced to two root causes: *apps under test require extra knowledge for effective testing*, and *tool defects prevent themselves from progressing*. We discuss specific issues w.r.t. these two root causes:

- **App logout or equivalent** (abbreviated as "LOUT" in Table 4.3) accounts for exploration tarpits on 23% (22 out of 96) of investigated regions with identified issues. Some apps essentially require login states for the majority of their functionalities to be accessible. However, the tools used in our experiments have no knowledge about the consequences of clicking the "logout" buttons in different apps before the tools actually try clicking these buttons. Unfortunately, after the tools try out such actions (driven by their exploration strategies), the apps' login states

Table 4.3: Distribution of confirmed issue types. Note that 'Mk' and 'Wt' refer to Monkey and WCTester, respectively. Issue type details are discussed in Section 4.5.2.

Issue	Rank-1				Rank-2				Rank-3				Total
	Ape	Mk	Wt	Sum	Ape	Mk	Wt	Sum	Ape	Mk	Wt	Sum	
LOUT	5	2	1	8	5	2	1	8	4	1	1	6	**22**
UI	0	4	0	4	0	3	0	3	2	4	0	6	**13**
NTWK	0	5	0	5	0	3	0	3	0	3	0	3	**11**
LOOP	0	0	7	7	0	0	8	8	0	0	5	5	**20**
ESC	0	4	0	4	0	2	0	2	0	2	0	2	**8**
ABS	0	0	2	2	0	0	2	2	0	0	1	1	**5**
MISC	5	1	2	8	5	1	0	6	3	0	0	3	**17**
Total	**10**	**16**	**12**	**38**	**10**	**11**	**11**	**32**	**9**	**10**	**7**	**26**	**96**

(both on-device and on-server) have been destroyed. The tools then have to spend all remaining time on a limited number of functionalities, leading to exploration tarpits. This case reveals a common weakness of existing UI test generation tools—they have a limited understanding of action semantics. As can be seen from Table 4.3, Ape is more likely to be affected by this type of issues.

- **Unresponsive UIs** ("UI") are found in 14% (13 out of 96) of regions. We find that some apps stop responding to UI actions after the advertisement banner is clicked, even though the apps' UI threads are not blocked. The issue is likely caused by the UI design defects in the Google AdMob SDK. The AUTs should be restarted as soon as possible to resume access to their functionalities. According to Table 4.3, Monkey is most vulnerable to such issues. One interesting finding is that Ape is actually also vulnerable to unresponsive UIs although the tool's implementation is capable of identifying such situations. However, while Ape proceeds to restart the app most of the times when Ape finds the app unresponsive, Ape fails to do so occasionally.

- **Network disconnections** ("NTWK") are found in 11% (11 out of 96) of regions. We find that turning networking off is undesirable for some apps, especially when the disconnection lasts for a long time. Consequently, these apps may show only messages prompting users to check their networks, leaving nothing for exploration. Tools such as Monkey can control network connections through Android system UI (e.g., by clicking the "Airplane Mode" icon). While the capability helps test app logic in edge conditions in general, it might hurt the tool's effectiveness on apps heavily relying on network access. All regions with the aforementioned issue come from Monkey's traces.

- **Restart/action loops** ("LOOP") are found in 21% (20 out of 96) of regions. The tool essentially keeps restarting or performing the same actions on the target app after some point. One potential cause for such issues is that the tool thinks that *all* UI elements in the target

app's main screen have been explored. The tool might need to revise its exploration strategy for discovering more explorable functionalities.

- **Obscure escapes** ("ESC") are found in 8% (8 out of 96) of regions. Defects in a tool's design or implementation can make it difficult for the tool to escape from certain app functionalities, and consequently, the tool loses opportunities to explore other functionalities. For instance, Monkey finds it challenging to escape a screen where the only exit is a tiny button on the screen (see the second motivating example in Section 4.2), due to the tool's lack of understanding of UI hierarchies.

- **UI abstraction defects** ("ABS") are found in 5% (5 out of 96) of regions. Defects in a tool's UI abstraction strategies can trick the tool into incorrectly understanding the testing progress. Seen from collected traces, WCTester considers all texts as part of abstract UI hierarchies. While the strategy works well in a wide range of testing scenarios, it keeps the tool repetitively triggering actions on UI elements with changing texts (such as counting down), given that the tool incorrectly thinks that new functionalities are being covered.

- **Miscellaneous tool implementation defects** ("MISC") are in 18% (17 out of 96) of regions. In our case, we find potential implementation defects in three tools: (1) Ape and Monkey fail to handle unresponsive apps ("Injection failed" or being unable to obtain UI hierarchies), and (2) WCTester is found to explore only a certain fraction of app functionalities after some point.

Another finding is that the ratio of confirmed issues decreases when the rank goes lower ($38/48 = 79\%$ for rank-1, $32/43 = 74\%$ for rank-2, and $26/40 = 65\%$ for rank-3), suggesting the usefulness of prioritizing regions based on their lengths.

4.5.3 RQ2. Improvement of Testing Performance

This section shows that the identified exploration tarpit regions by Vet can be used to automatically address tool issues.

Automatic fix application The essential idea is to prevent some tool issues from happening again or getting rid of tool issues quickly by controlling the interactions between tools and apps. Specifically, given an exploration tarpit region, we identify the UI element that the tool acts on right before the region begins, and then we use Toller to disable the UI element for Vet-guided runs. Many tool issues can be targeted by this simple approach. For example, if we disable the advertisement banners that lead to Unresponsive UIs in Section 4.5.2, tools will simply not run into the undesirable situation, and they can focus on testing other more valuable app functionalities. In some cases when there are multiple entries to the region and existing traces do not reveal all the entries, the aforementioned approach might fail. We mitigate this limitation by monitoring

and controlling the testing progress—currently, if we observe any of the most frequently appearing screens from Excessive Local Exploration regions, we restart the AUT in VET-guided runs.

We implement UI element disablement by relying on TOLLER to monitor screen changes during testing and dynamically modify UI element properties. When a target screen (i.e., a UI screen containing any target UI element, as determined by UI hierarchy equivalence check) shows up, we pinpoint the target UI element by matching with the path to each UI element from the root UI element. Once we confirm that the target UI element exists on the current screen, we instruct TOLLER to disable the UI element, which will not respond to any further action on itself. For the edge case where the action is not on any UI element (e.g., pressing the Back button), we restart the target app once we see the corresponding target screen. Note that there is no need to modify the app installation packages given that we manipulate app UIs dynamically.

Methodology of experiments We aim to measure the testing effectiveness throughout the entire process of applying VET. For each (tool, app) pair, in addition to the three *initial* runs for trace collection, we perform the experiments for three runs in each of these three settings: (1) Disabling UI elements based on rank-1 regions (rank-1 *guided* runs; see Section 4.4.1 for our ranking strategy), (2) Disabling UI element based on rank-1, rank-2, and rank-3 regions (rank-1/2/3 guided runs), and (3) Keep the same settings as initial runs (*comparison* runs). Note that each run also lasts for one hour, and all experiments are conducted in the same hardware and software environment regardless of different settings.

We measure method coverage (numbers of uniquely covered methods in app bytecode) as one testing effectiveness metric in our experiments. Note that methods involved by app initialization (i.e., before tools start to test) are excluded for a more precise comparison of code coverage gain. Upon each (tool, app) pair, we use the following Test Groups (TGs) for effectiveness comparison:

- *TG-1*: three initial runs and three comparison runs.
- *TG-2*: three initial runs and three rank-1 guided runs.
- *TG-3*: three initial runs and three rank-1/2/3 guided runs.

Each group consists of six one-hour runs, intended for reducing random biases. We accumulate the method coverage of all runs within a group for the group's method coverage. Test Group 1 serves as the baseline, while the other two test groups aim to measure how much testing effectiveness gain can VET users expect. The main reason for experimenting with exploration tarpit regions of different ranks is that there can be multiple tool issues on an AUT, and addressing only one of the issues might not suffice.

We also measure the crash triggering capabilities with cumulative numbers of distinct crashes. We consider only crashes from bytecode given that Android apps are predominantly written in

Table 4.4: Cumulative code coverage statistics. Note that '#M_n' shows the total number of covered methods in Test Group n. $\Delta M_n = (\#M_n - \#M_1) \div \#M_1 \times 100\%$.

App Name	Ape					Monkey					WCTester				
	$\#M_1$	Rank-1		Rank-1/2/3		$\#M_1$	Rank-1		Rank-1/2/3		$\#M_1$	Rank-1		Rank-1/2/3	
		$\#M_2$	ΔM_2	$\#M_3$	ΔM_3		$\#M_2$	ΔM_2	$\#M_3$	ΔM_3		$\#M_2$	ΔM_2	$\#M_3$	ΔM_3
AccuWeather	21977	22485	2.3%	22538	2.6%	14830	29266	97.3%	24990	68.5%	14982	15104	0.8%	14797	-1.2%
AutoScout24	17245	17274	0.2%	22713	31.7%	23455	23270	-0.8%	23085	-1.6%	18637	22157	18.9%	22154	18.9%
Duolingo	15186	15299	0.7%	15510	2.1%	13676	14598	6.7%	14582	6.6%	12319	14625	18.7%	14450	17.3%
Flipboard	9594	9742	1.5%	9510	-0.9%	5949	7482	25.8%	7125	19.8%	7872	7901	0.4%	8265	5.0%
Merriam-Webster	9056	9093	0.4%	9093	0.4%	5617	8992	60.1%	9275	65.1%	9328	9089	-2.6%	9010	-3.4%
Nike Run Club	30523	29937	-1.9%	29939	-1.9%	24472	24592	0.5%	26645	8.9%	19754	21936	11.0%	22460	13.7%
OneNote	6681	7134	6.8%	7028	5.2%	7131	7114	-0.2%	7381	3.5%	6453	6675	3.4%	6933	7.4%
Quizlet	17000	17114	0.7%	16900	-0.6%	13722	13995	2.0%	13995	2.0%	14679	14865	1.3%	14448	-1.6%
Spotify	19759	21475	8.7%	21475	8.7%	20616	22200	7.7%	21486	4.2%	18897	29298	55.0%	19632	3.9%
TripAdvisor	29857	30645	2.6%	31006	3.8%	16773	20919	24.7%	20919	24.7%	26773	28467	6.3%	28180	5.3%
trivago	20706	20710	0.0%	20711	0.0%	20216	20489	1.4%	20482	1.3%	19952	19964	0.1%	20032	0.4%
Wattpad	22960	22668	-1.3%	22447	-2.2%	14541	13717	-5.7%	15276	5.1%	15067	15884	5.4%	15982	6.1%
WEBTOON	32933	31674	-3.8%	31599	-4.1%	31477	30176	-4.1%	30176	-4.1%	25720	27659	7.5%	27659	7.5%
Wish	8829	8850	0.2%	9106	3.1%	8490	8522	0.4%	8269	-2.6%	6948	7191	3.5%	7207	3.7%
Youtube	26874	33301	23.9%	33757	25.6%	29316	32087	9.5%	35892	22.4%	22179	29143	31.4%	30233	36.3%
Zedge	42899	43074	0.4%	43433	1.2%	31245	38103	21.9%	44931	43.8%	36671	37464	2.2%	38343	4.6%
Average	**20755**	**21280**	**2.5%**	**21673**	**4.4%**	**17595**	**19720**	**12.1%**	**20282**	**15.3%**	**17264**	**19214**	**11.3%**	**18737**	**8.5%**

JVM languages (Java and Kotlin). Crashes are identified by hashing the code locations in stack traces. We additionally leverage Toller to disable each app's `UncaughtExceptionHandler`, which is widely used by industrial apps to collect crash reports and might prevent crash information from being exposed to the Android log system (i.e., Logcat [48]) and captured by our scripts.

It should also be noted that Toller monitors, captures, and manipulates UIs with negligible overheads; thus, the testing effectiveness of original runs should remain comparable with and without Toller in use. In addition, Vet analyzes traces very efficiently, usually requiring only a few seconds on a single trace, while analysis of multiple traces can be trivially parallelized.

Results Tables 4.4 and 4.5 show the effectiveness improvements by comparing three test groups. As can be seen from the results, automatically applying fixes based on Vet's identified exploration tarpit regions helps Ape, Monkey, and WCTester achieve up to 4.4%, 15.3%, and 11.3% cumulative code coverage improvements relatively on 16 apps using the same amount of time. Additionally, Vet helps Ape, Monkey, and WCTester achieve up to 2.1x, 2.1x, and 1.9x overall distinct crashes, respectively. It should be noted that Vet's automatic approach does not address all the tool issues—some issues, especially those rooting in tool implementations, are likely addressable by only humans.

For most (tool, app) pairs with improvements, considering only rank-1 exploration tarpit regions is sufficient for code coverage gain. However, there are cases where code coverage increases considerably when we consider rank top-3 regions instead. One explanation is that there are multiple applicability issues, or multiple instances of exploration tarpit corresponding to the same applicability issue. For example, when applying Monkey on the app Nike Run Club, there are multiple ways to enter a hard-to-escape functionality as depicted by the motivating example in Section 4.2. If we block only one entry, Monkey can still find other ways to enter the functionality

Table 4.5: Distinct crash statistics. Note that '#C$_n$' is for total # triggered unique crashes in Test Group n.

App Name	Ape			Monkey			WCTester		
	#C$_1$	#C$_2$	#C$_3$	#C$_1$	#C$_2$	#C$_3$	#C$_1$	#C$_2$	#C$_3$
AccuWeather	2	4	9	0	5	4	0	2	4
AutoScout24	1	1	1	2	1	1	0	0	0
Duolingo	1	2	2	0	0	0	1	1	1
Flipboard	0	0	1	0	0	1	1	1	0
Merriam-Webster	2	3	3	0	5	7	0	0	0
Nike Run Club	1	1	1	6	3	5	0	0	1
OneNote	0	2	5	0	2	1	0	1	0
Quizlet	0	1	0	0	0	0	1	1	1
Spotify	1	1	1	0	0	0	0	0	0
TripAdvisor	3	5	5	0	1	1	1	1	1
trivago	1	1	2	0	1	2	2	1	2
Wattpad	2	2	2	0	0	0	2	3	2
WEBTOON	1	1	1	1	0	0	0	3	3
Wish	2	4	2	2	1	1	0	0	0
Youtube	0	0	0	0	0	0	1	1	2
Zedge	0	0	0	0	0	0	0	0	0
Total	**17**	**28**	**35**	**11**	**19**	**23**	**9**	**15**	**17**

(despite being more difficult) and waste time there.

There are also cases where code coverage decreases when we consider rank-3 exploration tarpit regions. One reason is that lower-ranked regions might not capture real issues, but VET tries to "fix" them anyway, indeliberately interfering with normal functionalities. In the case of applying WCTester on Spotify, the rank-3 region does not reveal any tool issue, according to our observation. "Fixing" this region can cause VET to restart the app when one major functionality shows up. Consequently, WCTester is unable to explore that functionality to achieve more coverage in guided runs.

4.5.4 RQ3. Missed Tool Issues

We show our analysis of tool issues that are not revealed by any exploration tarpit regions (i.e., false negatives) reported by VET.

Methodology We propose using issue-specific detection tools to discover hidden tool issues (in all the collected traces), which can provide an estimation of how likely any issue is missed. Specifically, we summarize the characteristics of two issues from Section 4.5.2, *App logout* and *Unresponsive UIs*, to design two approaches specifically targeting these two issues. The reason for choosing the aforementioned issues is that (1) they have a substantial appearance among all issues

that we have identified, and (2) their existence is relatively more straightforward to be determined using our infrastructure. We do not adopt manual inspection due to the subjectivity and the error-prone nature of manual judgments, especially given that we need to look at all the collected traces entirely.

Our issue-specific detection approaches work as follows:

- *App logout.* We first manually look into the activity list of each app with 'Login' ticked in Table 4.2 and identify the subset of activities that are used for logging in to the app. Then, when we analyze a given trace, we find the first and the last occurrence of any activity that belongs to the aforementioned list. If there is any occurrence, and the time distance between the first and the last occurrence is at least $t_{\min}$, we regard that an issue of App logout is found in the trace.

- *Unresponsive UIs.* In our investigation, we find only one case that leads to Unresponsive UIs: when an advertisement banner is clicked. Consequently, a new activity can be observed, where the activity belongs to the Google AdMob SDK and has the same activity ID across different apps. Thus, we simply look for continuous appearance (i.e., there is no other activity in between) of the aforementioned activity ID on the given trace. Note that we require the appearance to be continuous so as to exclude the cases where the tool (such as Ape) chooses to restart the app. If the appearance lasts for at least $t_{\min}$, we regard that an issue of Unresponsive UIs is found in the trace.

In order for a detected issue to be considered covered by our general-purpose algorithms, we require that at least one algorithm-identified exploration tarpit region covers at least 1/2 of the time length within which the detected issue appears.

Results We apply the specialized approaches on all 144 collected traces. As a sanity check, we find that all the manually-discovered App logout and Unresponsive UIs issues are covered by the specialized approaches. We compare the results against identified regions from VET's general-purpose algorithms and perform manual confirmation. We find only several cases where the VET algorithms do not yield accurate results, discussed as follows:

- On one trace from applying Monkey on Zedge, VET misses one Unresponsive UIs issue by not reporting any covered region. We find that the Excessive Local Exploration algorithm prioritizes another region over any region covering this issue. However, we find that this issue is also present in other traces from the same (tool, app) pair, and VET identifies and addresses this issue.

- On each of two other traces from Ape on Duolingo and Monkey on Zedge, there are two instances of the same Unresponsive UIs issue, and VET reports only one of them. The inaccuracy is also caused by the prioritization strategy of the Excessive Local Exploration algorithm. However, since two instances point to the same issue on both traces, VET is still effective.

- On one trace from Ape on Quizlet, Ape logs out about only 5 minutes after testing starts, but VET reports only 22 minutes of exploration tarpit. We find Quizlet's UI design to be somewhat unique: the app has a special entry to some of the main functionalities in its landing page that is accessible without logging in. The entry is buried within a paragraph of texts, and the texts are shown only after a specific combination of swiping. Ape is able to find this special entry in this run, making VET confused. Nevertheless, VET is still able to find the correct trigger action from other traces.

4.6 DISCUSSION AND LIMITATIONS

We are mainly focusing on making VET useful in the context of automated UI testing. However, it should be noted that VET's potential usage scenarios are beyond automated UI testing. One usage case is for app UI quality assurance, where an app might have UI design issues with one or more functionalities. As a result, human users may face difficulties locating their desired features. When a human user runs into such situations, he/she will then likely search for the desired features through repeated (and ineffective) exploration around a few functionalities, and the difficulties can be reflected by the collected user behavior statistics. By subsequently utilizing VET's identification of exploration tarpits, we can quickly know which functionalities likely have the aforementioned UI design issues, potentially from numerous traces collected from end-users, and address these issues in a more timely manner.

One question is whether VET is capable of differentiating testing scenarios (1) that a tool is supposed to handle but does not (i.e., tool issues), and (2) that a tool is not expected to handle (i.e., beyond tool capabilities, such as apps requiring special inputs). We would like to point out that it is inherently difficult to differentiate these two types of scenarios due to lacking specifications over tool capabilities. On the other hand, VET can help users identify (and mitigate) the cases beyond a tool's capabilities, being already useful.

We acknowledge that TOLLER, the utility that monitors, captures, and manipulates UIs for VET, still has limitations. For example, the current implementation of TOLLER does not capture text inputs. However, adding support for text capturing is achievable with engineering efforts. Moreover, TOLLER's limitations do not prevent VET from being generalizable.

4.6.1 Comparison with Time-travel Testing

Recent work [70] proposes time-travel testing (referred to as TTT) to help Android UI testing escape from ineffective AUT states including loops and dead ends. TTT uses checkpoint and restore—checkpointing progressive states and restoring those states after loops and dead ends are detected.

VET is different from TTT in terms of design goals. First, TTT aims to recover from ineffective exploration, while VET mainly focuses on prevention. Second, exploration tarpits in VET are more general than TTT's lack-of-progress definitions. One example is logging out, where tools assisted by VET can still explore a fraction of app functionalities, such as registering and resetting passwords. Loops and dead ends are not necessarily present when exploring an app with only a few functionalities. Third, VET aims to enhance existing test generation tools without the need to understand their internal design or implementation, instead of building a new test generation tool that excels at all apps.

The design of VET brings a few advantages. First, as acknowledged by TTT [70], the state recovery may lead to inconsistent app states when testing apps with external state dependencies that are maintained at the server side. Note that controlling server-side states is challenging, e.g., many industrial apps use external services that the apps have no control of. VET's preventive strategy avoids this limitation. Second, VET's preventive strategy does not incur overhead for lack-of-progress detection or state recovery in guided runs. This strategy is specifically useful when a tool repeatedly gets into exploration tarpits. Third, VET does not require additional device support for state recovery (such as RAM data restoring).

4.7 THREATS TO VALIDITY

A major external threat to the validity of our work is the environmental dependencies of our subject apps. More specifically, many of the industrial apps in our experiments require networking for main functionalities to be usable, and it is possible for such dependency to change the behaviors of these apps despite our efforts to make our experiment environment consistent across different runs. In order to reduce the influences of environmental dependencies in our experiments, we repeat each experiment setting by three times and use aggregated metrics in our paper. We additionally control each tool's internal randomness by setting a constant random seed for each app. Nevertheless, this threat can be further reduced by involving more repetitions in our experiments.

A major internal threat to the validity of our work comes from the manual analysis of collected traces. We need to manually determine whether the exploration tarpit regions reported by VET indeed reveal any tool issue. Consequently, related evaluation results can be influenced by subjective judgments. However, it should be noted that any work involving manual judgments in the evaluation is vulnerable to this threat.

4.8 SUMMARY

We have exploited the opportunities of improving Android UI testing via automatically identifying and addressing exploration tarpits. Specifically, we have presented VET, a general approach

and supporting system for effectively identifying and addressing exploration tarpits. We have designed specialized algorithms to support VET's concepts. Our evaluation results have shown that VET identifies exploration tarpits that cost up to 98.6% of testing time budget, revealing various issues hindering testing efficacy. By trying to automatically fix the discovered issues, VET helps the Android UI test generation tools under evaluation with achieving up to 15.3% higher code coverage relatively and triggering up to 2.1x distinct crashes.

CHAPTER 5: EPIT: FACILITATING PARALLELIZATION COORDINATION WITH UI EXPLORATION SPACE PARTITIONING

5.1 OVERVIEW

Given the fast pace of app development and testing, app developers/testers can substantially benefit from parallelizing running a test generation tool on the AUT. Specifically, running a test generation tool on the AUT can be on multiple devices concurrently. [1] Compared with applying the tool on a single device, using the same amount of machine time, the parallelization allows comparable test effectiveness on the AUT using only a fraction of wait time, enabling the developers/testers to substantially shorten the testing duration of the AUT. The parallelization is especially desirable in the context of using testing clouds (offered by various industrial vendors [71, 72, 73, 74, 75]), where developers/testers have access to numerous cloud-hosted real devices and emulators while paying for only the duration when the devices in the cloud testing services are used (i.e., *machine time*) for testing their apps. This billing mode eliminates the need of investing high monetary costs to purchase and maintain multiple devices used for parallelized testing.

To maximize the aforementioned benefits, there is a strong need of a new approach to coordinate such parallelization for two main reasons. First, without coordination of parallelized test runs (i.e., running a given test generation tool against the AUT on multiple devices concurrently and independently), overlapped app explorations by the tool across devices tend to occur earlier during testing, causing more wasted machine time and testing budgets. The observation is reflected by our experimental results in Figure 5.4: the baseline suffers from slowed gain of new code coverage earlier than our approach. Second, existing UI test generation tools [8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 31] have been primarily focusing on the testing effectiveness of single test runs (i.e., running a given test generation tool against the AUT on a single device). It is desirable to design a new approach that can work seamlessly with these tools to the new context of parallelized testing, preferably in a manner of being applicable to any test generation tool, e.g., by only automatically manipulating the AUT.

To satisfy the preceding need, in this chapter, we propose Epit, a parallel testing approach that automatically manipulates the AUT to guide a UI test generation tool used during testing. Aiming to improve the overall test effectiveness or reduce the amount of needed machine time for comparable test effectiveness, Epit automatically partitions the AUT's UI exploration space and conceptually transforms the AUT into different variants that are suitable to be tested independently by the tool on different devices. Figure 5.1 shows an example of an app where we can divide the app into a main-functionality variant and an account-logistics variant that expose two very different sets of UIs and functionalities.

[1] In our work, we consider the situations where running a test generation tool with deterministic tool behaviors against the AUT on each of these devices exercises the same testing behaviors.

To realize Epit, partitioning the AUT's UI exploration space is challenging for three main reasons. First, loosely coupled UI subspaces often do not have explicit boundaries (e.g., specific UI elements that control the transitions among these UI subspaces) that can be recognized automatically despite being necessary for effective partition. Second, mistakenly partitioning two closely related UIs will likely make exploration less effective for the corresponding functionality. An example is the password input screen and the result feedback screen (by disabling the confirmation button) when exercising "Change password" as shown in Figure 5.1. Third, while it is possible for developers/testers to manually specify partition boundaries, the efforts are limited to a per-app basis and prone to functionality changes by app version iterations (note that most popular apps release multiple updates every month [76]), resulting in high manual efforts.

To tackle the challenge of partitioning UI spaces, Epit distributedly conducts our novel on-the-fly trace analysis during the testing process to find AUT UI subspaces with UIs that are strongly cohesive while being loosely coupled to other UIs. It is generally easier for the test generation tool to transition within a strongly cohesive UI subspace compared with transitioning between two loosely coupled UI subspaces, suggesting that the functionalities of two loosely coupled UI subspaces are not heavily dependent on each other. In particular, Epit conducts two algorithms (with details described later) to identify the AUT's UI subspaces on each test device on the fly, and then notifies all other devices to avoid entering the subspace by blocking the corresponding *entrypoint* (the action leading to the subspace), forcing different devices to discover and explore different loosely coupled UI subspaces.

To find loosely coupled UI subspaces, our on-the-fly trace analysis includes two algorithms: *Exploration Space Partition* and *Skewed Local Exploration*. First, Exploration Space Partition checks for the cases where, after some specific action, the fraction of AUT UI subspace explored by the tool is disjoint with most of the AUT UIs observed earlier. Second, Skewed Local Exploration captures the cases where, after some specific action, the tool enters a small AUT UI subspace, stays there for a substantial amount of time, and visits UIs in a highly skewed pattern (i.e., most visits are on only a few UIs). Both algorithms capture the aforementioned characteristics of loosely coupled UI subspaces. For the example shown in Figure 5.1, some time after the tool chooses to explore the account tab by clicking the account icon highlighted in Figure 5.1(a) on one device, Epit's trace analyzer reports that the tool has started exploring a different UI subspace around Figure 5.1(b) as opposed to Figure 5.1(a). The analyzer also reports that clicking the account icon causes the transition. Consequently, Epit instructs all the other devices to disable the account icon when Figure 5.1(a) shows up to force exploring other functionalities.

We evaluate Epit by experimenting on three state-of-the-art tools (namely Monkey [7], Ape [15], and WCTester [32, 33]) and 16 popular industrial apps (as shown in Table 5.1) from recent work [23]. We find that on average Epit helps Monkey, Ape, and WCTester reach comparable code coverage using 85%, 32%, and 85% less machine time w.r.t. the baseline. When giving the same amount of machine time, we find that on average Epit helps Monkey, Ape, and WCTester

achieve 24.9%, 3.4%, and 16.2% more cumulative code coverage relative to the baseline, plus 2.9×, 1.1×, and 1.3× distinct crashes. When measuring the degree of overlapped explorations, we find that on average Epit helps Monkey, Ape, and WCTester reduce the numbers of occurrences of distinct UIs by 71.1%, 44.8%, and 62.8%, respectively. These results indicate Epit's high value in terms of improving the test effectiveness or reducing the cost of testing by reducing overlapped explorations.

In summary, this chapter makes the following main contributions:

- An approach to improve the overall effectiveness of parallelized UI testing of running a test generation tool based on our novel on-the-fly trace analysis;
- A practical system applicable to any existing Android UI test generation tool on any app;
- A comprehensive evaluation of Epit, showing that Epit helps existing test generation tools substantially reduce overlapped explorations and achieve comparable code coverage using up to 85% less machine time than the baseline.

5.2 MOTIVATING EXAMPLE

In this section, we explain loosely coupled UI subspaces in Android apps with an illustrative example.

Figure 5.1 presents the account tab in the app "Quizlet". As the app's main functionalities, the app aims to help users learn through specialized assistance and flashcards, which can be accessed from the home screen shown in Figure 5.1(a). The app requires users to register accounts and stay logged in to access its services. Meanwhile, the app allows users to personalize their accounts, such as changing the profile photo or the account password, in the account tab as shown in Figure 5.1(b) (note that our account information is blacked out in the screenshot for anonymization). Changing these settings will generally not affect how users find or interact with the app's main functionalities. Users can activate the account tab by clicking the account icon in the bottom right corner of the home screen as shown in Figure 5.1(a).

In this example, the subspace of UIs accessed when testing the app's main functionalities is loosely coupled with the subspace corresponding to UIs accessible in the account tab. Due to the fact that the test generation tool needs specific sequences of actions to go back to the home screen and switch to the account tab when exploring the app's main functionalities (such as flashcards), it is generally more difficult to transition between the main-functionality UI subspace and the account-tab UI subspace compared with transitioning within one UI subspace, suggesting that the functionalities of these two UI subspaces are not heavily dependent on each other. Such patterns of transitions can be recognized by our on-the-fly trace analysis algorithms. Consequently, when we conduct parallelized UI testing for the aforementioned app, we can manipulate the entry of the

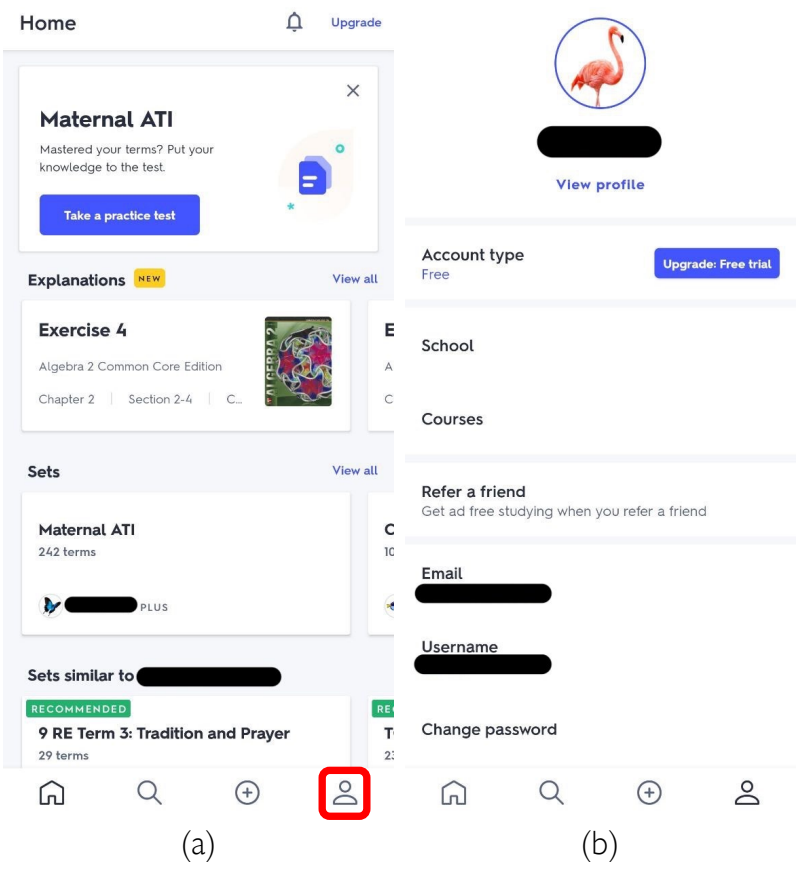


Figure 5.1: A motivating example: Account tab in the app "Quizlet".

account tab (e.g., by disabling clicking the account icon) to control which part of functionalities can be exercised on two devices and let them exercise different functionalities.

5.3 THE EPIT APPROACH

In this section, we present the design details of EPIT.

5.3.1 Mechanism Overview

We first present a high-level overview of EPIT's mechanism. Within a *parallel run* conducted by EPIT, multiple *partial runs* are performed by applying the given UI test generation tool on the App Under Test (AUT) for certain amounts of time without interruption. EPIT aims to identify loosely coupled UI subspaces by monitoring each partial run and control access to these UI subspaces by manipulating the *entrypoints* (specific UI actions that cause transitions to these UI subspaces) in all partial runs. Note that our assumptions are that the original test generation tool in one partial run does not have mechanism to communicate with the tool in another partial run, and that the tool's exploration strategy is driven by randomness from the tool or the AUT

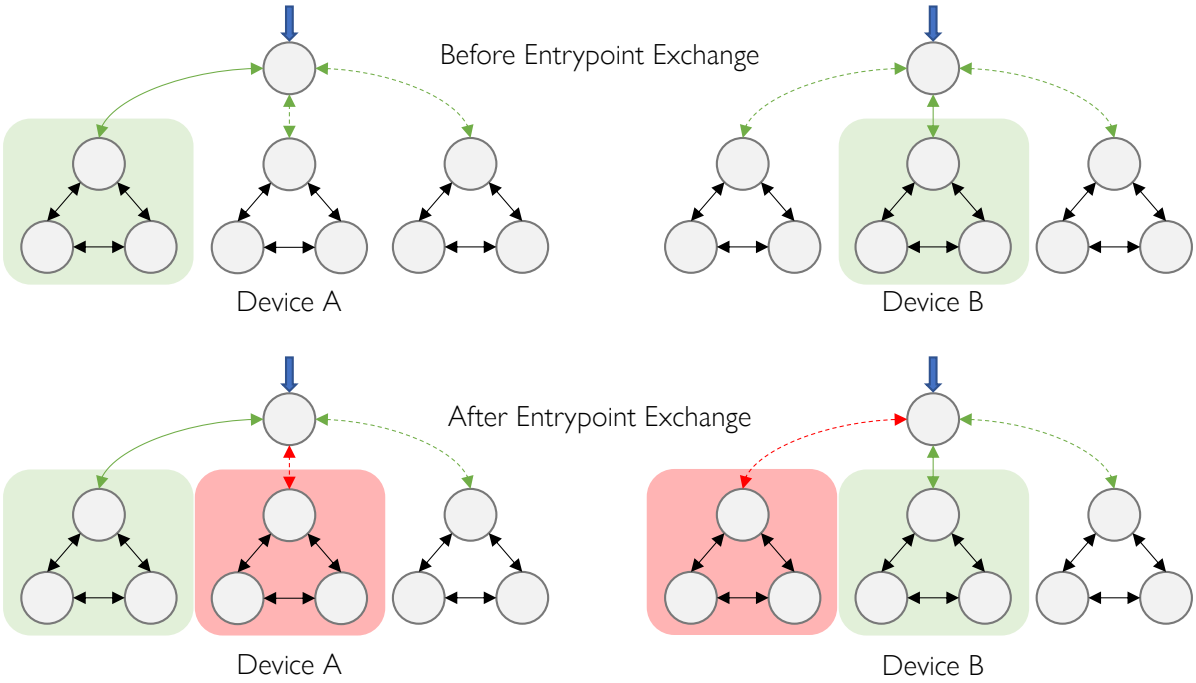


Figure 5.2: Sharing UI subspace entrypoint information across devices. Note that gray circles indicate UI screens, which formulate loosely coupled UI subspaces. Arrows show possible transitions among UI screens, where colored arrows (green or red) represent entrypoints to loosely coupled UI subspaces. A green arrow indicates that the transition is allowed, while a red arrow indicates that the entrypoint is blocked and the tool is not supposed to enter the corresponding UI subspace (with a red background). Solid colored arrows show that the tool has already explored the corresponding UI subspaces (with green backgrounds) through the entrypoints. Dotted colored arrows indicate that the tool has not passed the corresponding entrypoints.

(true in most situations).

Figure 5.2 illustrates a case when there are two partial runs being performed concurrently on two devices on an AUT with three loosely coupled UI subspaces. At the beginning of the parallel run, no loosely coupled UI subspace has been identified yet, and the test generation tool is allowed to explore any UI subspace of the AUT in each partial run. Then, as indicated by the upper half of Figure 5.2, the tool gets into subspace 1 on device A and subspace 2 on device B and explores the UIs within the subspaces. After some time, through monitoring and on-the-fly trace analysis, Epit acknowledges that subspace 1 and 2 have already been exercised on device A and B, respectively. Epit also figures out the entrypoints to subspace 1 and 2 with help from trace analysis algorithms. Then, Epit instructs the test monitors on device A and B to block entrypoints to subspace 2 and 1, respectively. In the end, as shown by the lower half of Figure 5.2, the tool is unable to explore subspace 2 and 1 on device A and B, forcing itself to explore other globally unvisited parts of AUT UIs (e.g., subspace 3). The aforementioned process is continuously performed throughout the parallel run, and different partial runs are expected to cover various sets of AUT functionalities.

Algorithm 5.1: FindPartition: Find the exploration space partition point inside a given list of UIs

Input: List of UIs S, list of timestamps of UIs T, the minimum time length after partition $l_{\min}$
Output: Index of the found partition point, or nil

```
N, p_out, score_min ← |S|, −1, 1
foreach p ∈ N − 1 downto 0 do
    if T[p] ⩽ T[N − 1] − l_min then
        p_max ← p
        break
    end
end
sample_size ← |Set(S[p_max + 1 : N])|
foreach p ∈ 1 to p_max do
    overlap_size ← 0
    foreach s ∈ Set(S[0 : p]) do
        overlap_size ← overlap_size + CountIn(s, S[p : N])
    end
    overlap_score ← overlap_size / (N − p)
    purity_score ← Sigmoid(|Set(S[p:N])| / sample_size − 1)
    score ← overlap_score + 2 * purity_score − 1
    if score < score_min then
        score_min ← score
        p_out ← p
    end
end
if |Set(S[0 : p_out])| > |Set(S[p_out : N])| then
    return p_out
else
    return nil
end
```

5.3.2 On-the-fly Trace Analysis

We present the details of two on-the-fly trace analysis algorithms adapted from [23] to capture loosely coupled UI subspaces. A *trace* is defined as a sequence of UIs interleaving with actions on them. Each algorithm can be divided into two parts: (1) a partition/trapping point finder (Algorithm 5.1 and 5.2) that works on a given list of UIs and tells which UI indicates the beginning of Exploration Space Partition/Skewed Local Exploration, which indicates the existence of a newly-found loosely coupled UI subspace, and (2) an on-the-fly detection wrapper (Algorithm 5.3) that invokes the finder upon receiving newly-observed UIs and confirms and reports the partition/trapping point along the way. Epit is then able to infer the entrypoint from the trace based on the partition/trapping point.

FindPartition (Algorithm 5.1), reused from [23], aims to find the proper partition point such

Algorithm 5.2: FindTrapped: Find the exploration space trapping point inside a given list of UIs

Input: List of UIs S, list of timestamps of UIs T, the minimum time length after trapping $l_{\min}$
Output: Index of the found trapping point, or nil
$N \leftarrow |S|$
$p_{\text{out}}, score_{\min} \leftarrow -1, 1$
$R \leftarrow \text{Merge}(S)$ // $R : S \mapsto P$ s.t. $P \subseteq S$
foreach $p \in 0$ to $N-1$ **do**
  $S' \leftarrow \text{Map}(S[p:N], R)$
  $freq \leftarrow []$
  foreach $s \in \text{Set}(S')$ **do**
    $freq \leftarrow freq :: \text{CountIn}(s, S')$
  end
  SortDescendingly($freq$)
  $skewness \leftarrow 1$
  foreach $i \in 1$ to $|freq| - 1$ **do**
    $skewness \leftarrow skewness + \frac{1}{\log(\frac{freq[i-1]}{freq[i]})+1} * \frac{1}{i}$
  end
  $score \leftarrow \frac{|S'|}{T[N-1]-T[p]} * skewness$
  if $score < score_{min}$ **then**
    $score_{\min} \leftarrow score$
    $p_{\text{out}} \leftarrow p$
  end
end
if $T[N-1] - T[p_{out}] \geqslant l_{min}$ **then**
  return p_{out}
else
  return nil
end

that the observed UI subspaces at two sides are barely relevant (line 10-14) and overlapped explorations becomes substantial afterwards (line 2-8 and 15). The goal aligns with our desired characteristics of loosely coupled UI subspaces. A sanity check on whether the observed UI subspace becomes smaller after partition (line 22) helps Epit choose larger UI subspaces to further decompose, given that it usually helps to keep parallel tasks evenly distributed.

FindTrapped (Algorithm 5.2), enhanced from the Excessive Local Exploration algorithm from [23], aims to find the proper trapping point, after which the tool is observed to get stuck within a small UI subspace while visiting UIs in a highly skewed pattern for a long time (line 15). The goal also aligns with our desired characteristics of loosely coupled UI subspaces. The skewness is characterized by the relative frequency of each UI being observed (line 6-14). We favor the situations where very few UIs are visited much more often than others. We also reuse the Merge algorithm and UI hierarchy abstraction strategy from [23] to address the issue of UI

Algorithm 5.3: StreamFindPartition/Trapped: Find and confirm the exploration space partition/trapping point given a stream of UIs, intended for on-the-fly detection

Input: A newly-seen UI s, represented by its abstract UI hierarchy
Output: UI of the partition/trapping point, or nil if no such point is found and confirmed
State: List of seen UIs S, list of timestamps of UIs T, last confirmed point p_{last}, timestamp of last p_{last} value change t_p
Const: Wait time length before starting to find l_{wait}, wait time length before confirming p_{last} as a partition/trapping point l_{conf}

```
t ← CurrentTimestamp()
S, T ← S :: s, T :: t
if t − T[0] ⩾ l_wait then
    p ← FindPartition(S, l_wait)    // Or FindTrapped(S, l_wait)
    if p != p_last then
        p_last, t_p ← p, t
    else if p != nil ∧ t − t_p ⩾ l_conf then
        return S[p]
    end
end
return nil
```

variants (line 3).

StreamFindPartition/Trapped (Algorithm 5.3) accomplishes on-the-fly detection by continuously invoking FindPartition/FindTrapped and checking whether the detection procedure produces consistent output for a given amount of time (t_{conf}). The intuition is that if the detection procedure does not provide a confident output due to targeted patterns not matching the inputs, the output is likely to change given more inputs.

5.3.3 System Design

Figure 5.3 depicts the overall system design of Epit. Specifically,

Trace analyzer monitors the traces collected on each device using the Toller framework [22] (which monitors and reports immediately any action along with the context AUT UI) and invokes on-the-fly detection algorithms to identify entrypoints to UI subspaces that the test generation tool has already explored on the current device. Aiming to balance the detection accuracy and efficiency, we run two instances of each detection algorithm using two groups of parameters: $G_1 = \{l_{\text{wait}} = l_{\text{conf}} = 5 \text{ minutes}\}$, and $G_2 = \{l_{\text{wait}} = 1 \text{ minute}, l_{\text{conf}} = 0.5 \text{ minutes}\}$. Outputs of the two groups of algorithms are treated differently as explained later.

Test coordinator, upon receiving the entrypoints reported by trace analyzers, decides whether to distribute the information to all devices (to block these entrypoints) and whether to allocate or de-allocate devices. Specifically,

- Entrypoints reported by G_1 are broadcasted to all devices immediately. The strategy aims to

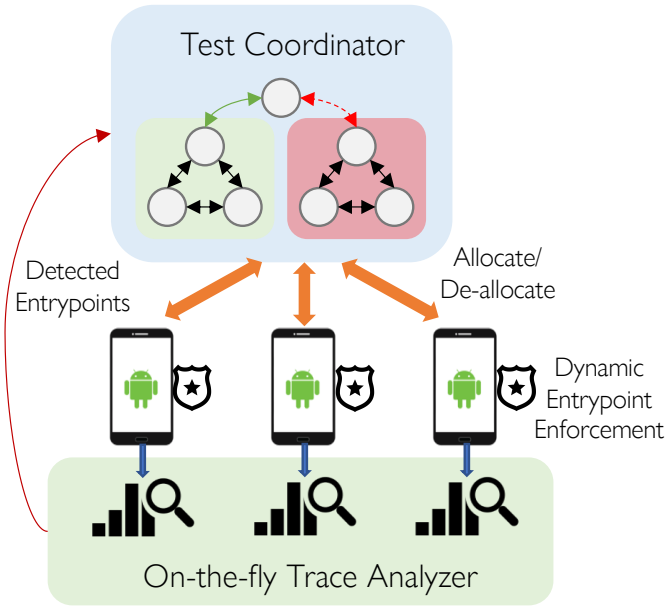


Figure 5.3: Overall system design of Epit

alleviate the situation where the test generation tool is unable to escape from the UI subspace captured by the algorithms. Additionally, for each entrypoint reported by Exploration Space Partition, Epit will de-allocate the source device after 10 minutes and allocate a new device (after re-initializing the testing environment and applying existing entrypoint blocking information) and start a new partial run for the remaining test time budget. For each entrypoint reported by Skewed Local Exploration, Epit will restart the AUT on the source device.

- Each entrypoint reported by G_2 is not broadcasted immediately but rather held for confirmation. The strategy aims to improve the accuracy of faster detection. Once there is another report on the same entrypoint (from any device), the entrypoint will be broadcasted. No other actions will be taken.

Dynamic entrypoint enforcement constantly monitors the current screen contents on each device and takes actions to block entrypoints broadcasted by the test coordinator. More specifically, Epit relies on Toller for notifications of screen updates. Upon each screen update, a UI hierarchy is taken, from which Epit identifies UI elements that match any blocked entrypoint. Epit then instructs Toller to disable these UI elements before the test generation tool has chances to action on them.

Epit supports device allocation constrained by either *machine time* (corresponding to the amount of computing power used) or *wait time* (time elapsed after all partial runs are finished). More specifically,

- Algorithm 5.4 shows how device allocation and de-allocation are handled in a machine-time constrained parallel run. The user needs to set l_m, l_p, and $d_{\max}$ (line 2). The test coordinator will

Algorithm 5.4: Handling device allocation and de-allocation for machine-time constrained parallel runs

State: Used machine time l_u, number of running devices d_r
Const: Total amount of machine time available l_m, maximum test time budget for each partial run l_p, maximum amount of devices allowed to run concurrently $d_{\max}$
Procedure CheckAllocateNewDevice()
  if $l_u < l_m \wedge d_r < d_{max}$ **then**
    $test\_time \leftarrow \min(l_m - l_u, l_p)$
    $l_u, d_r \leftarrow l_u + test\_time, d_r + 1$
    AllocateDeviceForPartialRun($test\_time$)
  end
Procedure OnDeallocateDevice*(remain_test_time)*
  $l_u, d_r \leftarrow l_u - remain\_test\_time, d_r - 1$
  CheckAllocateNewDevice()

initially launch one device to conduct one partial run by calling CheckAllocateNewDevice. When there is a new entrypoint available for broadcasting, CheckAllocateNewDevice will also be called. When a device gets de-allocated (by either timing out or being stopped by Exploration Space Partition), OnDeallocateDevice is called to report any machine time that has not been consumed. When all device have been de-allocated, the parallel run is finished.

- In a wait-time constrained parallel run, the user needs to set the number of devices to run concurrently $d_{\max}$ as well as the test time budget for each device l_p. Then the test coordinator will simply launch $d_{\max}$ devices to perform partial runs concurrently, each with timeout of l_p. Exploration Space Partition may request device re-allocations during the parallel run, but the total amount of test time on each device will remain constant.

As can be seen, a machine-time constrained parallel run requires more wait time to complete compared with a wait-time constrained parallel run with the same amount of l_p and machine time. However, machine-time constrained parallel runs should generally help tools further reduce overlapped explorations and achieve better overall test effectiveness, thanks to the reduced probabilities of two partial runs getting into the same UI subspaces (which is unavoidable when partial runs are executed concurrently given the delayed feedback of trace analysis). We provide the option for developers/testers to choose how to balance between wait time and overall test effectiveness.

5.4 EVALUATION

Our evaluation answers the following research questions:

- **RQ1:** Can Epit help reduce machine time needed to achieve comparable code coverage with the baseline?
- **RQ2:** Can Epit help improve the effectiveness of existing tools with parallel testing, measured by cumulative code coverage and unique numbers of crashes?
- **RQ3:** Can Epit help reduce overlapped explorations across partial runs, compared with the baseline?
- **RQ4:** How much wait time does Epit need to conduct machine-time constrained parallel runs?

5.4.1 Evaluation Setup

UI test generation tools and subject apps. Three state-of-the-art Android UI test generation tools are involved in our evaluation: Monkey [7], Ape [15], and WCTester [32, 33]. We use 16 popular industrial-quality Android apps from the Google Play Store [21] (see Table 5.1 for details). These apps are from a previous study [20] that compares the effectiveness of multiple tools over the most popular apps from different categories on Google Play. We choose only apps that work properly on our testing infrastructure, specifically: (1) they need to run on our x64 Android emulators (especially when they have native libraries), and (2) Toller is able to obtain UI hierarchies in most of the functionalities of these apps. Additionally, we skip apps that (1) have very limited sets of functionalities, or (2) require logging in for access to most features but we are unable to obtain a consistently usable test account.

Test platform. All experiments are conducted on the official Android x64 emulators running Android 6.0 on a server with Xeon E5-2650 v4 processors. Each emulator is allocated with 4 CPU cores, 2 GB RAM, and 2 GB internal storage. Emulator data are stored on an in-memory disk for minimal mutual influences caused by disk I/O bottlenecks. Hardware graphics acceleration is also enabled to ensure the responsiveness of emulators. We manually write auto-login scripts for apps with "Login" ticked in Table 5.1. Each of these scripts is executed only once before the corresponding app starts to be tested in each partial run.

Coverage and crash collection. We collect the method coverage as code coverage achieved by each partial run, using the MiniTrace [55] tool from Ape. By modifying DalvikVM/ART, the tool does not require app instrumentation, avoiding unexpected issues from modifying industrial apps in our experiments. Note that we exclude methods that are already covered after setting up the test environment but before the test generation tool starts to work in each partial run. As for crashes, we consider only those originated from apps' bytecode. Code locations in stack traces are used to identify unique crashes. We obtain stack traces by monitoring Android Logcat [48] messages. We also use Toller to remove apps' `UncaughtExceptionHandler`s to ensure that all stack traces are exposed to Logcat instead of being suppressed by apps' own crash handlers.

Table 5.1: Overview of industrial apps used for evaluation. Note that '#Install' denotes the approximate number of downloads. 'Login' indicates whether the app requires logging in to access most features.

App Name	Version	Category	#Install	Login
AccuWeather	7.4.1-5	Weather	50m+	✗
AutoScout24	9.8.6	Auto & Vehicles	10m+	✗
Duolingo	3.75.1	Education	100m+	✗
Flipboard	4.1.1	News & Magazines	500m+	✓
Merriam-Webster	4.1.2	Books & Reference	10m+	✗
Nike Run Club	2.14.1	Health & Fitness	10m+	✓
OneNote	16.0.9126	Business	100m+	✓
Quizlet	6.6.2	Education	10m+	✓
Spotify	8.4.48	Music & Audio	100m+	✓
TripAdvisor	25.6.1	Food & Drink	100m+	✗
trivago	4.9.4	Travel & Local	10m+	✗
Wattpad	6.82.0	Books & Reference	100m+	✓
WEBTOON	2.4.3	Comics	10m+	✗
Wish	22.5.0	Shopping	100m+	✓
YouTube	15.35.42	Video Player & Editor	1b+	✗
Zedge	7.34.4	Personalization	100m+	✗

Parallel run settings. We conduct three parallel runs on each combination of test generation tools and apps with three settings: baseline, Epit with constrained machine time, and Epit with constrained wait time. We set $l_p = 1$ hour and $d_{\max} = 5$ in all settings, and $l_m = l_p \times d_{\max} = 5$ machine hours when running Epit with constrained machine time. For each *baseline run*, we simply start $d_{\max}$ test devices at once and let the test generation tool explore the AUT for l_p without interruption or interference. As can be seen, each parallel run is allocated with five machine hours.

5.4.2 RQ1. Reduction of needed machine time

For this RQ, we study how Epit can benefit parallelized UI testing by investigating how many computing resources (measure in machine time) Epit can help save if we hope to reach the same test effectiveness as the baseline. We use the cumulative code coverage (averaged across all apps) as the test effectiveness metric and aim to find out how the metric changes along with elapsed machine time, on all three test generation tools in both machine-time and wait-time constrained parallel runs.

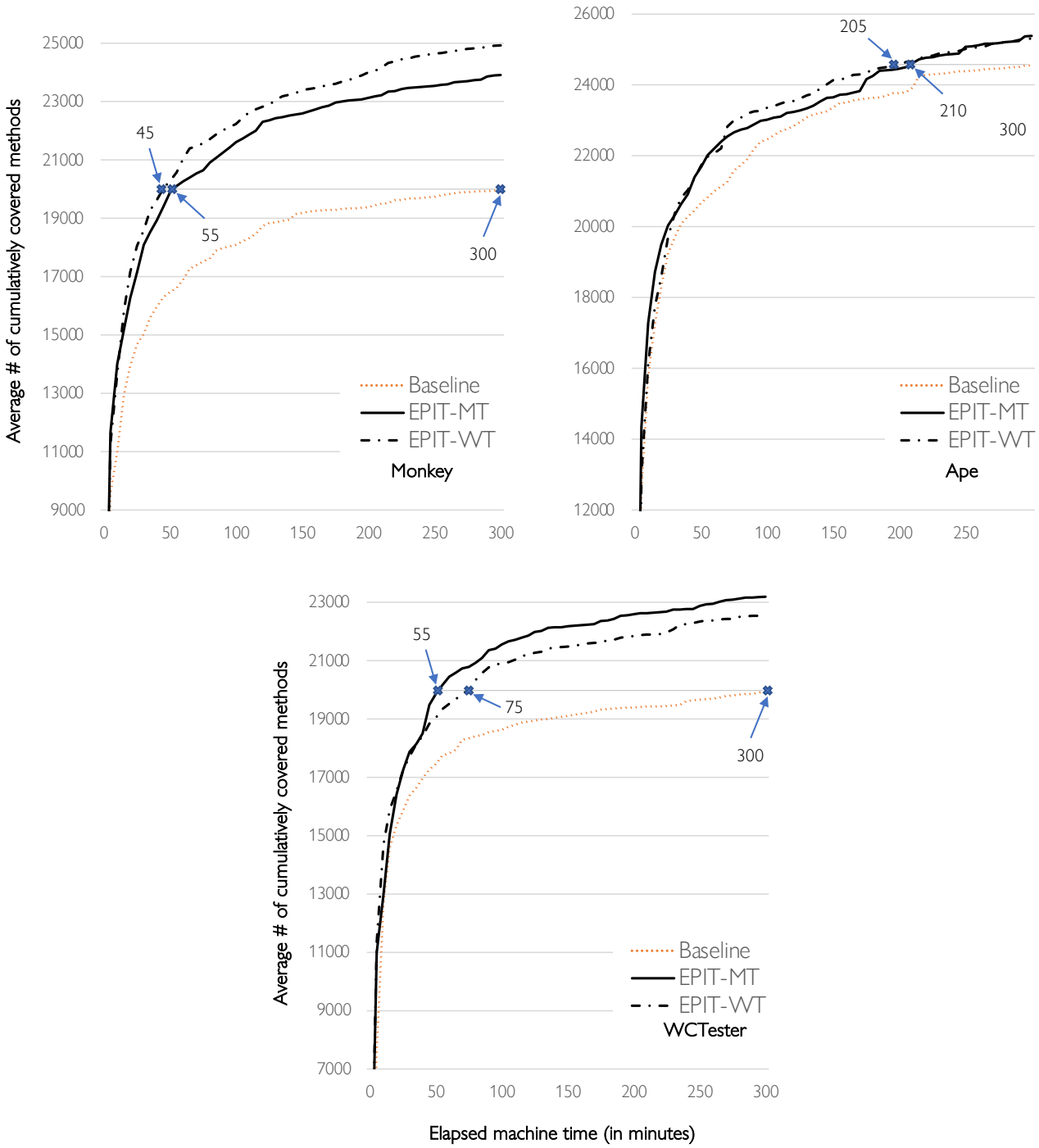


Figure 5.4: Trend of average cumulative code coverage by elapsed machine time during parallel runs. Note that each data point shows how many methods have been covered on average across all apps by the respective tool, after the corresponding amount of machine time has elapsed in each parallel run. A method is regarded as being covered by the tool in a parallel run if the method is covered in any partial run. "Epit-MT" and "Epit-WT" refer to machine-time/wait-time constrained parallel runs conducted by Epit. To show the machine time saved by Epit, in each sub-figure we add a horizontal line that crosses the end of baseline's curve and intersects with other two curves. Each intersection point is marked with an 'X' as well as the corresponding machine time reading.

Table 5.2: Cumulative code coverage statistics. Note that for the corresponding tool $\Delta = (\#\textbf{Epit} - \#\textbf{Baseline}) \div \#\textbf{Baseline} \times 100\%$.

App Name	Baseline			Epit (machine time)			Epit (wait time)		
	monkey	ape	wctester	monkey	ape	wctester	monkey	ape	wctester
AccuWeather	13064	27102	30296	27836	26854	16593	26919	27261	22163
AutoScout24	34518	40230	35637	39861	40473	37865	39509	39622	36796
Duolingo	11908	15264	14512	14601	15131	14542	14577	15169	14212
Flipboard	6113	11822	6781	7760	11041	9199	7755	11504	8135
Merriam-Webster	8319	9696	9378	10525	10914	10508	8683	9311	8614
Nike Run Club	21089	27135	17004	21582	26696	21607	36347	26689	39872
OneNote	6854	6847	6520	7696	6848	6165	7509	7111	6841
Quizlet	40294	44466	35085	46454	45943	38631	44882	46724	33462
Spotify	20163	18249	14377	21274	21928	18542	22553	19207	16940
TripAdvisor	16469	27981	25407	22504	29678	28647	24433	30780	26052
trivago	19980	20397	19988	20487	20350	20335	20088	20509	20093
Wattpad	15021	21857	14419	16490	22152	18352	22193	21895	14345
WEBTOON	25448	27603	22434	31566	30104	28514	27930	28009	27224
Wish	20690	27021	12725	22268	30152	21685	24827	33536	20806
Youtube	26050	28574	27557	33875	29706	29072	33975	29431	26707
Zedge	33449	38652	27098	37892	38256	50785	36661	38188	38303
Average	**19964**	**24556**	**19951**	**23917**	**25389**	**23190**	**24928**	**25309**	**22535**
Δ	-	-	-	**19.8%**	**3.4%**	**16.2%**	**24.9%**	**3.1%**	**13.0%**

Figure 5.4 shows the trend of average cumulative code coverage by elapsed machine time during parallel runs, where data points are available every five machine minutes. From the figure, we find that Epit-conducted parallel runs use substantially less machine time to achieve comparable average code coverage with the baseline runs, with 82%, 30%, 82% fewer, and 85%, 32%, 75% fewer machine time needed by Monkey, Ape, WCTester in machine-time and wait-time constrained parallel runs, respectively. When these numbers are translated into costs, developers/testers can save up to 85% of prices they pay for testing their mobile apps. Considering that computing resources are still expensive for mobile testing (e.g., AWS Device Farm charges $0.17 per device minute for real devices [71]), involving Epit in parallelized UI testing can bring about substantial economical benefits.

5.4.3 RQ2. Test effectiveness improvement

This RQ aims to find out whether Epit is able to help test generation tools achieve better test effectiveness given the same amount of machine time. We compare the cumulative code coverage and numbers of distinct crashes of all parallel runs on each combination of test generation tools and apps. For each parallel run, its cumulative code coverage / distinct crashes are calculated as the union of distinct methods covered / crashes triggered in each partial run.

Table 5.2 shows the statistics of cumulative code coverage on all tools. As can be seen, Epit-conducted parallel runs generally achieve higher code coverage compared with the baseline runs, with 19.8%, 3.4%, 16.2% more, and 24.9%, 3.1%, 13.0% more methods covered by Monkey, Ape,

Table 5.3: Distinct crash statistics.

App Name	Baseline			Epit (machine time)			Epit (wait time)		
	monkey	ape	wctester	monkey	ape	wctester	monkey	ape	wctester
AccuWeather	0	1	0	3	1	1	2	2	0
AutoScout24	0	0	0	0	0	0	0	0	0
Duolingo	0	4	1	0	4	1	0	1	1
Flipboard	0	1	2	2	0	2	0	0	1
Merriam-Webster	0	1	0	6	3	1	5	1	0
Nike Run Club	4	0	1	4	1	1	2	1	0
OneNote	0	0	0	3	0	0	0	0	0
Quizlet	4	3	0	6	2	0	3	2	0
Spotify	0	0	0	0	0	0	0	0	0
TripAdvisor	0	3	3	0	2	0	0	4	2
trivago	1	0	0	2	0	0	1	0	0
Wattpad	0	0	1	0	0	3	3	0	1
WEBTOON	1	1	0	0	1	0	2	1	0
Wish	0	0	0	3	0	0	0	0	0
Youtube	0	0	2	0	2	3	1	1	4
Zedge	0	0	0	0	0	1	0	0	0
Total	**10**	**14**	**10**	**29**	**16**	**13**	**19**	**13**	**9**

WCTester on average in machine-time and wait-time constrained parallel runs, respectively. Meanwhile, Table 5.3 shows the statistics of distinct crashes on all tools. While machine-time constrained parallel runs achieve more crashes than the baseline runs on all tools (2.9×, 1.1×, 1.3× by Monkey, Ape, WCTester, respectively), wait-time constrained parallel runs have difficulties outperforming the baseline runs on Ape and WCTester despite being able to do so on Monkey.

The different outcomes of two time-constraining strategies are expected as discussed in Section 5.3.3. Essentially, machine-time constrained parallel runs trade wait time for better test effectiveness. Section 5.4.5 discusses how much extra wait time is needed by the machine-time constrained parallel runs. Another observation is that Epit helps Monkey achieve even higher code coverage in wait-time constrained parallel runs compared with machine-time constrained parallel runs. A possible explanation is that Monkey encounters many more crashes in machine-time constrained parallel runs (as shown in Table 5.3), making it more difficult for the tool to explore deeper AUT functionalities.

5.4.4 RQ3. Reduction of overlapped explorations

This RQ aims to find out whether Epit is capable of reducing overlapped explorations of AUT functionalities within parallel runs. To show the degree of overlapped explorations within a parallel run, we propose a new metric by measuring the average number of occurrences of distinct UIs observed during testing across all partial runs. We represent UIs by their abstract UI hierarchies (using the strategy from [23]) to avoid being overly sensitive to screen content changes.

Table 5.4: UI overlap statistics, measured by the average number of occurrences of distinct UIs. Note that for the corresponding tool $\Delta = (\#\textbf{Baseline} - \#\textbf{Epit}) \div \#\textbf{Baseline} \times 100\%$.

App Name	Baseline			Epit (machine time)			Epit (wait time)		
	monkey	ape	wctester	monkey	ape	wctester	monkey	ape	wctester
AccuWeather	31.3	15.7	13.7	2.9	7.7	15.9	2.8	5.4	20.7
AutoScout24	18.6	7.4	18.3	8.8	6.2	15.5	6.5	6.9	12.5
Duolingo	35.6	27.2	21.6	21.3	17.1	20.3	21.9	18.9	22.0
Flipboard	16.2	14.6	8.8	6.4	9.9	9.0	6.0	10.8	8.0
Merriam-Webster	14.2	51.5	16.4	6.9	44.4	35.9	7.2	32.0	36.4
Nike Run Club	36.9	20.6	135.7	6.7	16.9	21.7	8.8	14.6	19.8
OneNote	19.7	106.1	29.6	7.6	21.5	35.2	12.4	23.5	32.2
Quizlet	13.3	8.0	17.3	6.8	7.8	12.5	5.9	6.6	16.3
Spotify	17.8	14.9	27.2	7.3	10.9	13.2	7.5	9.9	14.1
TripAdvisor	62.6	6.9	6.9	7.2	7.2	8.8	11.3	6.9	7.6
trivago	13.4	8.6	11.8	4.0	6.1	10.7	4.7	6.1	14.1
Wattpad	10.1	5.0	7.1	6.5	4.8	6.0	8.0	4.7	6.9
WEBTOON	37.9	14.2	14.8	13.9	11.0	15.4	14.4	12.7	15.1
Wish	21.0	22.0	11.3	6.3	9.3	9.6	6.4	8.5	8.7
Youtube	7.2	4.8	4.3	3.4	4.3	4.0	4.0	4.2	3.9
Zedge	87.7	21.6	320.4	11.9	18.2	14.0	18.7	21.0	62.3
Average	**27.7**	**21.8**	**41.6**	**8.0**	**12.7**	**15.5**	**9.2**	**12.0**	**18.8**
Δ	-	-	-	**71.1%**	**41.7%**	**62.8%**	**66.9%**	**44.8%**	**54.8%**

Table 5.4 shows the statistics of overlapped UIs for three parallel runs on each combination of test generation tools and apps. As can be seen, Epit-conducted parallel runs have substantially smaller UI overlaps on average compared with the baseline runs, with 71.1%, 41.7%, 62.8% fewer, and 66.9%, 44.8%, 54.8% fewer per-UI occurrences by Monkey, Ape, WCTester in machine-time and wait-time constrained parallel runs, respectively.

One finding is that machine-time constrained parallel runs have smaller UI overlaps compared with wait-time constrained parallel runs on Monkey and WCTester (as expected given the discussions in Section 5.3.3) while having slightly larger UI overlaps on Ape. The unexpected difference is mainly caused by a dictionary app "Merriam-Webster", on which Ape explores distinct UIs for many more times in the machine-time constrained parallel run. We investigate the test logs and find that the app's functionalities exercised by Ape are rather concentrated (as indicated by the high UI occurrence numbers compared with other apps), and our algorithms are confused when determining the UI subspace boundaries. For instance, after more than one machine hour spent in the machine-time constrained parallel run, the Exploration Space Partition algorithm determines that the word search box should be disabled, making it more difficult to reach the word definition screen, contributing to UI overlapping afterwards. However, it should be noted that Epit still helps Ape improve the test effectiveness on this app, for example by discovering and blocking entrypoints to the ads screen and keeping Ape focused on the app's main functionalities.

5.4.5 RQ4. Needed wait time of machine-time constrained parallel runs

As discussed in Section 5.3.3, machine-time constrained parallel runs provide an option to re-balance between wait time and overall test effectiveness. While previous RQs show that machine-time constrained parallel runs generally achieve better overall test effectiveness compared with wait-time constrained parallel runs, it is necessary to know how much wait time is actually needed by machine-time constrained parallel runs. We answer this RQ by showing the distribution of wait time of each machine-time constrained parallel run on different apps by each test generation tool.

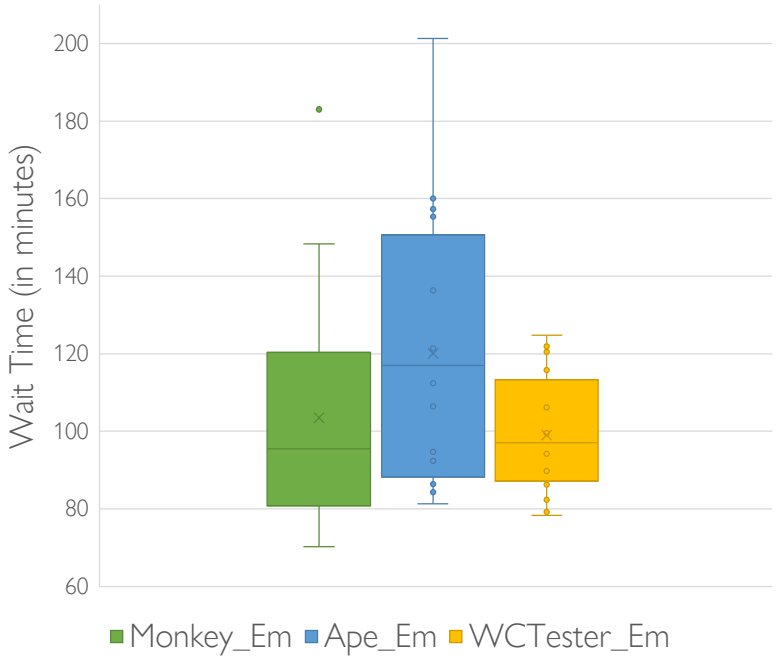


Figure 5.5: Distribution of needed wait time for machine-time constrained parallel runs by Epit

Figure 5.5 shows the aforementioned distribution using box plots. Note that each wait-time constrained parallel run uses 60 minutes of wait time, and machine-time constrained parallel runs are expected to use more (up to 300 minutes in the worst case, when all partial runs execute sequentially). As can be seen, the amount of needed wait time varies for different tools, with Monkey and WCTester sharing a similar median number (96 and 97 minutes to be exact), while Ape runs generally last longer (with a median of 117 minutes). The results suggest that it often takes more time for Epit's detection algorithms to confirm entrypoints in Ape runs, likely caused by Ape's strategy to prioritize unvisited UIs globally (recall that Epit's detection algorithms with shorter confirmation periods need to report the same entrypoint twice before actions can be taken, as discussed in Section 5.3.3). Ultimately, we think it will be up to developers/testers to decide whether the trade-off between wait time and overall test effectiveness is sufficiently valuable.

Implications of dynamic analysis. Due to the fact that we rely on dynamic analysis on UI traces to find loosely coupled UI subspaces, it is possible that Epit does not find all possible loosely coupled UI subspaces that can be exercised effectively by parallel testing. However, we insist on dynamic analysis instead of conducting static analysis to obtain the loosely coupled UI subspaces on industrial apps for two main reasons. First, static analysis has scalability issues on industrial-quality apps as used in our evaluation; the large code base of industrial apps (millions of lines of code) makes existing static analysis tend to fail on them and most industrial apps adopt obfuscation techniques [77] against static analysis. Second, even if we can find some loosely coupled UI subspaces by static analysis, it can be very challenging to find the feasible input sequences to get to these subspaces (e.g., submitting a payment), and consequently these statically detected UI subspaces do not contribute to the improvement of parallel testing effectiveness.

Another issue of dynamic analysis is that, in some edge cases, Epit's on-the-fly detection algorithms might incorrectly mark loosely coupled UI subspaces. Specifically, test generation tools occasionally have internal issues that cause the tools to get stuck at certain screens, likely due to implementation defects. At most time, the tool can overpass this issue after restarting or leaving these screens. However, in this case, the on-the-fly detection algorithm may incorrectly determine that the tool is exploring a strongly cohesive UI subspace. We argue that Epit is still useful in these cases, in the sense that Epit can proactively help tools avoid UIs/functionalities that trigger their defects.

Multiple entrypoints to an identical UI subspace. Our entrypoint blocking mechanism conservatively blocks the entrypoint (i.e., the UI widget) that previously led to the UI subspace that we want to block access to, if it appears in the current test run. It is possible for our entrypoint blocking mechanism to be ineffective if there are multiple entrypoints to the identical UI subspace that we hope to block access to.

This limitation can be alleviated by developing a mechanism for escaping from a given UI subspace, but the escaping mechanism heavily depends on the structure of the UI subspace. First, escaping the given UI subspace may require substantial efforts to restore the state of apps. Second, different entrypoints suggest that there are different contexts to exercise the given UI subspace, and the semantics of actions (e.g., which page to jump back to after signing up for an account) could change. It may be context-sensitive whether entering a given UI subspace from different entrypoints exercises different functionalities. We plan to study on the existence and influence of multiple entrypoints and model on whether allowing different entrypoints contribute to the testing effectiveness.

Uneven partitioning of UI space. Given the incompleteness of the identified loosely coupled UI subspaces from dynamic analysis, it is possible for Epit to partition an AUT's UI space unevenly for different devices. This issue can be addressed using *recursive partitioning*. Specifically,

when we find new loosely coupled UI subspaces whose entrypoints have been indirectly blocked by other previously identified entrypoints, we can temporarily unblock these previously identified entrypoints so that other devices will have chances to further explore the UI subspaces inadvertently blocked by these entrypoints. However, it needs non-trivial design to determine whether an entrypoint relies on another, as well as how long a previously identified entrypoint should be unblocked to avoid more overlapped explorations.

Relationship with FastBot. FastBot [78] works by first transferring GUI information and actions of an AUT to a directed graphical model. The tool then applies reinforcement learning techniques to generate test inputs that maximize the state coverage of the model. As claimed in its paper, FastBot supports testing on multiple devices in parallel. However, the parallel testing conducted by FastBot heavily relies on the reinforcement learning algorithm and cannot be applied to an arbitrary tool as Epit does. In addition, the parallel testing capability of FastBot is not publicly available for reproducing due to lack of implementation details. Consequently, we do not involve FastBot in our evaluation.

5.6 THREATS TO VALIDITY

There might be both internal and external threats to the validity of our work.

The main external threat comes from the environmental dependencies of our subject apps. To be specific, part of our subject apps require network access to maintain the main functionalities. Even though we try to ensure the consistency of our experimental environment during the experiment process, such network dependencies still have uncertainty and may affect the performance of the relevant apps in the experiment. Towards minimizing the effects of such environmental dependencies, we make each parallel run include five partial runs and use aggregated metrics.

The major internal threat to our work would be the potential faults from both the implementation of Toller's and the setup of all Android test generation tools involved in our experiments, which may affect our experiment results. Moreover, the scripts for table and figure generation may include incorrectness and affect our results. To mitigate these internal threats to the validity of our work, we output the relevant logs and the used metrics for each experiment. Furthermore, to ensure that the generated results match our observations, we apply manual inspection to analyze the sample of the experiment logs.

5.7 SUMMARY

In this chapter, we have looked into the opportunities of improving the overall testing effectiveness or reducing testing costs in the context of parallelized UI testing. Specifically, we have presented Epit, a parallel testing approach that automatically manipulates the AUT to guide the

UI test generation tool used during testing. Based on two algorithms, namely *Exploration Space Partition* and *Skewed Local Exploration*, Epit conducts on-the-fly trace analysis during the testing process to find loosely coupled AUT UI subspaces and coordinate the devices to explore these UI subspaces. Conceptually, Epit transforms the AUT into different variants suitable to be tested independently by the tool on different devices. To evaluate Epit, we have applied it on 16 popular industrial apps with three state-of-the-art tools for automated Android UI test generation. Our evaluation results have shown that Epit helps these tools reach comparable code coverage using up to 85% less machine time than the baseline, and Epit helps reduce overlapped explorations by up to 71.1%.

CHAPTER 6: RELATED WORK

Automated UI Testing for Android. This dissertation describes work on enhancing existing automated mobile UI test generation tools. A number of automated UI test generation tools have been published over the years of development. One of the earliest efforts is Monkey [7], a tool from Google and shipped with nearly every Android device, originally intended for stress testing of app UIs. While receiving almost no feedback from the target app, Monkey still manages to achieve relatively good test effectiveness with its high event generation/execution efficiency and wide range of supported UI event types. Many sophisticated automatic Android UI test generation tools have been developed after Monkey, mainly focusing on novel exploration algorithms. Such tools can be generally divided into three categories based on their exploration algorithms:

Randomness/evolution based [8, 9, 10]. For instance, Sapienz [8] is an evolutionary-testing-based test generation tool for Android UI testing. It leverages a genetic algorithm [42] to evolve generated seed input sequences to search for the optimized test suites containing short input sequences while maximizing code coverage and fault revelation.

UI model based [11, 12, 13, 14, 15, 32, 33, 70, 78]. A UI model is essentially a UI transition graph associated with information useful for exploration planning. The model is mainly useful for determining which AUT UIs have been explored so far as well as planning on UI input sequences to reach a specific target. To put UI modelling into action, SwiftHand [11] designs an active learning algorithm that minimizes app restarts while exploring the AUT and builds a deterministic UI model that approximates the AUT's UI transitions. Stoat [13] instead aims to construct a stochastic model of UI transitions from the AUT, mainly using dynamic analysis. The tool first constructs an initial model through random crawling. The tool then iteratively mutates and refines the model by generating and experimenting with UI input sequences sampled from the model on the AUT, while aiming to achieve high code/model coverage and exhibit diverse UI input sequences. FastBot [78] also aims to construct a UI model using dynamic analysis, but the tool uses reinforcement learning techniques to generate UI input sequences that maximize the state coverage of the model.

Systematic exploration based [16, 17, 18]. ACTEve [18] uses concolic testing to generate UI input sequences for efficient coverage of AUT code while addressing the issue of path explosion by pruning subsumption among different UI input sequences. A^3E [16] includes a systematic test generation 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 AUT 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 AUTs' bytecode.

There are also other perspectives on enhancing the exploration algorithms. Mao et al. [41] propose combining pre-defined human-generated input sequences (i.e., motif genes) with Sapienz's random exploration strategy to provide local exercise for different types of UI widgets. Sun et al. [79] propose modifying Android system settings during testing to find related defects in AUTs that cannot be triggered previously. He et al. [80] propose enhancing a tool's text input generation capability with constraint solving on pre-defined categories of textual hints from AUT UIs, allowing the tool to reach deeper AUT functionalities.

While existing work has been focusing on improving code coverage and triggering more crashes, there is also exploration on detecting non-crashing logic defects in AUTs. Su et al. [81] propose Genie, a fully automated approach for functional fuzzing of Android apps. Inspired by metamorphic testing, Genie leverages the pre-defined yet commonly-held rules to check whether the AUT behavior is consistent across generated UI inputs that are not supposed to cause the AUT to break the rules. The approach achieves a reasonable true positive rate of 40.9% in the evaluation involving 12 real-world Android apps.

UI Capture and Replay for Android. UI capture and replay is closely related to automated UI testing. Our work Toller also directly benefits many Android UI capture and replay tools that often require UI Hierarchy Capturing and UI Event Execution. Existing Android UI capture and replay tools can be categorized into two groups based on their level of understanding of AUT UIs:

Raw input based [82, 83, 84, 85, 86, 87]. This group of tools do not attempt to understand the structures of AUT UIs. Instead, these tools faithfully record the raw inputs (e.g., coordinates of clicking) from the test device and try to replay these inputs later, making them inherently fragile to nondeterminism and changing environments [24]. RERAN [85] is a relatively simple yet highly efficient tool of this group. Running on a computer, the tool solely relies on ADB shell commands (e.g., `getevent`) to record and replay input events on a rooted device. Culebra [87], by providing a remote control interface for the test device and letting users action on this interface, eliminates the requirement of the test device being rooted. However, the highly inefficient implementation prevents the tool from being practically usable [24]. appetizer-toolkit [82], on the other hand, solves the issue of root access and inefficiency with its proprietary design and implementation. The tool is additionally capable of handling changing environments in certain conditions (e.g., replaying on a different device with the same screen aspect ratio). There are also efforts [88, 89] to replay user inputs from videos so that the capturing process can be least intrusive. The effectiveness of these techniques is still limited by the capabilities and performance of deep learning.

UI element based [53, 84, 90, 91, 92, 93, 94]. This group of tools understand the structures of AUT UIs and perform UI element-level matching during replaying to cope with nondeterminism and changing environments, at the expense of efficiency and compatibility with more apps [24]. SARA and RANDR [93, 94] are state-of-the-art tools of this group. During recording, SARA relies

on its self-replay technique for good user experiences, essentially first recording raw inputs from the user and then converting the raw inputs into UI element-aware input sequences by automatically replaying on the same device. On the other hand, RANDR obtains UI element-aware inputs directly by instrumenting the AUT and asking the user to action on the instrumented AUT. Both tools feature the cross-device replaying capability.

Infrastructure Support for Android Testing. There has been work trying to improve infrastructure support for the purpose of efficient testing, similar to what Toller (as described in this dissertation) aims to achieve. Hu et al. [95] propose work that aims to quickly find potential sequences of error-triggering UI inputs through direct invocations of UI event handlers, achieved by instrumenting the target apps. Song et al. [96] also propose a similar idea. Toller's UI Event Execution support achieves similar goals. It should be noted that Toller aims to provide infrastructure support for any tool in need of either UI Hierarchy Capturing or UI Event Execution. Additionally, Toller does not require app instrumentation, which often breaks functionalities, especially on industrial apps.

Non-intrusive UI Testing. While it is usually convenient, accurate, and efficient to obtain UI information from and inject UI actions on the test device using system APIs (such as Android UIAutomator [29] and our work Toller), sometimes developers/testers do not have access to such interfaces, for instance, when testing IoT devices. Besides, developers/testers may hope to make their testing system applicable to different device platforms (e.g., Android and iOS). Performing UI testing in a non-intrusive way (i.e., using exactly the same interfaces as human users) is very helpful in such cases. There have been various efforts in this direction:

Computer Vision (CV) for UI testing. UI test generation tools might sometimes have access to only screenshots of test devices, where involving CV techniques is necessary. There exists a range of work leveraging CV techniques in software testing [97, 98, 99, 100, 101, 102]. Sikuli [97, 103] proposes a visual approach that enables developers/testers to write test scripts that use images to describe UI elements, making it easier to facilitate test automation. JAutomate [98] proposes a visual capture-and-replay technique by combining image recognition with capture and replay functionality. Choudhary et al. [99] and He et al. [100, 101] use computer vision techniques for cross-browser compatibility testing. White et al. [104] build a deep-learning model for widget detection from screenshots to automatically test open-source Java-based desktop applications.

Robotic testing. Robotic arms [105, 106, 107, 108] can be used to mimic how humans interact with physical devices in the situations where there is no programmatic interface to inject UI actions. Dhanapal et al. [105] use robotic arms for hardware functionality and performance testing of smart devices. Qian et al. [107] build a robotic testing system for IoT devices using the capture-and-replay technique to generate visual test scripts from videos.

Trace and Log Analysis. Our work (VET and EPIT) involves trace analysis. Existing work leverages trace and log analysis for various purposes, including: (1) Anomaly detection [109, 110, 111, 112]. LogRobust [109] converts unstable logs (e.g., those containing previously unseen log events or sequences) into sequential semantic vectors and feeds them to an attention-based Bi-LSTM model, which captures the context in the log sequences and learns the importance of different log events automatically; (2) Cause locating [113, 114, 115, 116, 117]. Kairux [113] assists locating the root cause of faults in sophisticated distributed systems by automatically comparing the traces from both failure and non-failure executions, where the first step at which the failure execution deviates from the non-failure execution with the longest common sequence prefix is regarded as exhibiting the root cause; (3) Failure reproduction [118]. Pensieve [118] reproduces failures by generating event chains of the failure through trace analysis. Specifically, Pensieve first captures the partial trace of events that occur during the fault execution process. The tool then iteratively analyzes and detects dependent events that may be relevant to the failure; (4) Performance-issue detection [119, 120]. Lprof [119] non-intrusively reconstructs the execution flow of each request from runtime logs to help developers understand performance issues of a distributed application, achieved by using static program analysis to automatically infer how the log should be parsed to extract useful information. Note that VET and EPIT focus on UI traces, which are generally different from logs and traces produced by or intended for troubleshooting program code.

Parallel Testing. Our work EPIT is related to parallel testing in the sense of conceptually producing multiple variants of the target program (i.e., customizing the AUT by manipulating the UI entries) for the test generation tool to work on. Existing software parallel testing work focuses on different components in the testing process. Mateo et al. [121] present a study of parallel mutation testing, where mutants and tests are executed in parallel processors to reduce the total time needed to perform mutation analysis. Jones et al. [122] introduce parallel debugging to help developers efficiently cope with multiple faults in a program, achieved by proposing an automated technique that partitions the set of failing test cases into clusters that target different faults. Staats et al. [123] propose a technique named Simple Static Partitioning to effectively partition the tasks of exploring different program paths for multiple computing nodes for efficient symbolic execution, achieved by performing a shallow symbolic execution on the target program in the upfront to compute the pre-condition for each node to follow. Bucur et al. [124] propose Cloud9, a platform for scalable parallelization of symbolic execution on large computing clusters. Note that existing parallel testing techniques that involve program analysis, mainly designed for traditional programs, will likely face added difficulties when applied to rich-GUI software programs as what we focus on. The main reason is that the control and data dependencies among application UIs are usually hidden deeply within program logic and states that control the UIs. Consequently, reasoning over the underlying program for information about UIs will likely be exponentially more expensive compared with doing so on traditional programs.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

Given the prosperity of mobile apps and the ever-growing complexity and fast pace of app feature development, there are unprecedented challenges on making these mobile apps robust and reliable. Automated UI testing, by mimicking how human users interact with apps through the UIs to detect reliability and usability issues with little human intervention, is gaining popularity as a complementary approach to manual and scripted testing. However, despite over a decade of research mainly focusing on designing novel exploration algorithms, it is still challenging for mobile UI test generation tools to achieve satisfactory effectiveness, especially on industrial apps with rich features and large code bases (Chapter 2).

In the context of existing work's heavy focus on designing novel exploration algorithms, this dissertation presented three parts of research that explore the possibilities for *existing* automated UI test generation tools to be empowered with *external* automated support. These parts of work enhance different components in the workflow of automated Android UI test generation tools: (1) *Infrastructure support* (Chapter 3), which enables a tool's exploration algorithm to obtain states from and execute actions on the test device, allows the tool to iterate faster and cover more App Under Test (AUT) functionalities within limited time; (2) *Exploration guidance* (Chapter 4), based on our observation that a tool's exploration algorithm or implementation might have applicability issues on a specific AUT in certain conditions; (3) *Parallelization coordination* (Chapter 5), for a tool and a specific AUT on multiple test devices, improves the overall test effectiveness or reduce testing costs by reducing overlapped explorations.

For infrastructure support, this dissertation presented Toller, a tool consisting of efficient mechanism for two types of UI operations (UI Hierarchy Capturing and UI Event Execution) through infrastructure enhancements to the Android operating system. Toller injects itself into the same virtual machine as the app under test, giving Toller direct access to the app's runtime memory. Toller is thus able to directly (1) access UI data structures, and thus capture contents on the screen without the overhead of invoking the Android framework services or remote procedure calls (RPCs), and (2) invoke UI event handlers without needing to execute the UI events. Compared with the often-used UIAutomator [29], Toller reduces average time usage of UI Hierarchy Capturing and UI Event Execution operations by up to 97% and 95%, respectively. We integrate Toller with existing state-of-the-art Android UI test generation tools and achieve the range of 11.8% to 70.1% relative code coverage improvement on average. We also find that Toller-enhanced tools are able to trigger 1.4x to 3.6x distinct crashes compared with their original versions without Toller enhancement.

For exploration guidance, this dissertation presented Vet, a general approach and its supporting system to automatically identify and resolve exploration tarpits for the given specific Android UI test generation tool on the given specific AUT. Vet runs the tool on the AUT for some time and

records UI traces, based on which VET identifies exploration tarpits by recognizing their patterns in the UI traces. VET then pinpoints the actions (e.g., clicking logout) or the screens that lead to or exhibit exploration tarpits. In subsequent test runs, VET guides the test generation tool to prevent or recover from exploration tarpits. From our evaluation with state-of-the-art Android UI test generation tools on popular industrial apps, VET identifies exploration tarpits that cost up to 98.6% testing time budget. These exploration tarpits reveal not only limitations in UI exploration strategies but also defects in tool implementations. VET automatically addresses the identified exploration tarpits, enabling each evaluated tool to achieve higher code coverage and improve crash-triggering capabilities.

For parallelization coordination, this dissertation presented EPIT, a parallel testing approach that automatically manipulates the AUT to guide the UI test generation tool used during testing, to improve the overall test effectiveness or reduce testing costs. EPIT conducts our novel on-the-fly trace analysis during the testing process to find loosely coupled AUT UI subspaces desirable for partitioning. By controlling access to these UI subspaces during testing, EPIT conceptually transforms the AUT into different variants suitable to be tested independently by the tool on different devices. Our evaluation results show that EPIT helps state-of-the-art tools reach comparable code coverage using up to 85% less machine time than the baseline. In addition, EPIT helps reduce the overlapped explorations by up to 71.1%.

7.1 FUTURE WORK

In this section, I discuss several potential directions to further stretch the work from this dissertation.

Involving domain knowledge from general large language models (LLMs) in automated UI testing. Domain knowledge can be very helpful for automated UI testing in certain cases. For example, generating text inputs that satisfy the AUT's validation rules can help the UI test generation tool unlock more AUT functionalities. There have been efforts [41, 80, 125] trying to involve domain knowledge in automated UI testing, achieved either manually or by building specialized models, limiting the generalizability and scope of application. With recent substantial progress on LLMs such as GPT-3 [126] and Codex [127], it is interesting to see whether a general LLM can be used to provide domain knowledge for UI test generation tools in a unified and elegant way. For instance, we can construct textual hints from AUT UIs and provide these hints as code comments for Codex to synthesize realistic text inputs or even generators of such text inputs.

Reproducibility of crashes triggered by automated UI test generation tools. Reproducing AUT crashes makes it easier for developers/testers to diagnose and troubleshoot their

apps. However, as researchers and practitioners have been focusing on the test effectiveness of automated UI test generation tools, it generally remains unknown how easily the crashes triggered by these tools can be reproduced. Our experience with existing tools suggests that reproducing such crashes is likely a challenging problem, mainly due to the need of replaying long UI input sequences generated by tools. Specifically, as long as one action in the sequence is incorrectly replayed, the corresponding crash might not be reproducible. Furthermore, input sequence replayability can be complicated by the widely existing nondeterminism in complex industrial apps (e.g., due to network access). Given that existing capture and replay tools are mainly intended for relatively short and slow-paced human-generated input sequences, these tools will likely fall short on the new demand of replaying tool-generated input sequences. Potential solutions for this challenge include pruning/minimizing the crash-triggering input sequences [1] and targeted searching for the crash (with the known crash-triggering input sequence as reference, e.g., by reusing sub-sequences).

Controlling the testing environment to amplify automated UI testing. In the real world, an AUT's behavior is usually not solely decided by the UI inputs. The AUT often needs to interact with the testing environment (such as the test device operating system and remote servers), which can also change the AUT's behavior and reveal AUT defects that are not triggerable using only UI inputs. Based on this insight, test amplification [128, 129, 130] has been proposed and applied on scripted Android UI testing to extend existing test cases' coverage on AUT code and find more defects. It would be interesting to see how test amplification can help with automated UI testing. While Su et al. [79] have explored augmenting Android system settings during automated UI testing to find relevant AUT defects, there are other environmental factors to consider. For example, controlling the AUT's networking (both states and contents) through mocking can be very helpful, given that most popular mobile apps nowadays require network access to provide services. Existing work on cloud API testing [131, 132, 133, 134] might provide useful techniques for this direction.

Transferring testing-related domain knowledge to other apps in automated UI testing. Existing work has been focusing on testing individual apps. In the real world, automated UI testing is sometimes conducted on many different apps in a centralized manner, e.g., by testing service providers. Meanwhile, industrial apps (especially those of the same category) often share UI designs for similar functionalities, e.g., account management. It would be interesting to see whether such similarities can help automated UI test generation tools get adapted more quickly to new AUTs based on the history of testing, achieved by transferring testing-related domain knowledge. One perspective is on the knowledge of exploration tarpits. For instance, after we learn from testing one app that clicking "logout" early in the test will likely lead to an exploration tarpit, we can avoid doing so on other apps without having to go through the process of trace collection and analysis, saving substantial amount of time.

REFERENCES

[1] W. Choi, K. Sen, G. Necula, and W. Wang, "DetReduce: Minimizing Android GUI Test Suites for Regression Testing," in *ICSE*, 2018.

[2] S. T. Help, "Why Mobile Testing Is Tough?" 2022. [Online]. Available: https://www.softwaretestinghelp.com/why-mobile-testing-is-tough/

[3] M. Labs, "10 Best Android & iOS Automation App Testing Tools," 2018. [Online]. Available: https://mobilelabsinc.com/blog/top-10-automated-testing-tools-for-mobile-app-testing

[4] Testsigma, "How to perform Mobile Automation Testing of the UI?" 2021. [Online]. Available: https://testsigma.com/blog/how-to-perform-mobile-automation-testing-of-the-ui/

[5] MoQuality, "How to Automate Mobile App Testing," 2021. [Online]. Available: https://www.moquality.com/blog/How-to-Automate-Mobile-App-Testing

[6] SmartBear, "Mastering the Art of Mobile Testing," 2021. [Online]. Available: https://smartbear.com/resources/ebooks/mastering-the-art-of-mobile-testing/

[7] Google, "Android Monkey," 2021. [Online]. Available: https://developer.android.com/studio/test/monkey

[8] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-Objective Automated Testing for Android Applications," in *ISSTA*, 2016.

[9] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *ESEC/FSE*, 2013.

[10] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag," in *MoMM*, 2013.

[11] W. Choi, G. Necula, and K. Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in *OOPSLA*, 2013.

[12] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps," in *MobiSys*, 2014.

[13] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, Stochastic Model-based GUI Testing of Android Apps," in *ESEC/FSE*, 2017.

[14] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A Lightweight UI-guided Test Input Generator for Android," in *ICSE-C*, 2017.

[15] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI Testing of Android Applications via Model Abstraction and Refinement," in *ICSE*, 2019.

[16] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps," in *OOPSLA*, 2013.

[17] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented Evolutionary Testing of Android Apps," in *FSE*, 2014.

[18] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *FSE*, 2012.

[19] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" in *ASE*, 2015.

[20] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An Empirical Study of Android Test Generation Tools in Industrial Cases," in *ASE*, 2018.

[21] Google, "Android Apps on Play Store," 2018. [Online]. Available: https://play.google.com/store/apps

[22] W. Wang, W. Lam, and T. Xie, "An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools," in *ISSTA*, 2021.

[23] W. Wang, W. Yang, T. Xu, and T. Xie, "Vet: Identifying and Avoiding UI Exploration Tarpits," in *ESEC/FSE*, 2021.

[24] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie, "Record and Replay for Android: Are We There Yet in Industrial Cases?" in *ESEC/FSE*, 2017.

[25] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie, "A Large-Scale Empirical Study on Android Runtime-Permission Rationale Messages," in *VL/HCC*, 2018.

[26] B. Andow, S. Y. Mahmud, W. Wang, J. Whitaker, W. Enck, B. Reaves, K. Singh, and T. Xie, "PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play," in *USENIX Security*, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/andow

[27] W. Wang, W. Zheng, D. Liu, C. Zhang, Q. Zeng, Y. Deng, W. Yang, P. He, and T. Xie, "Detecting Failures of Neural Machine Translation in the Absence of Reference Translations," in *DSN Industry Track*, 2019.

[28] W. Zheng, W. Wang, D. Liu, C. Zhang, Q. Zeng, Y. Deng, W. Yang, P. He, and T. Xie, "Testing Untestable Neural Machine Translation: An Industrial Case," in *ICSE-Companion*, 2019.

[29] Google, "UI Automator," 2020. [Online]. Available: https://developer.android.com/training/testing/ui-automator

[30] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," in *ASE*, 2012.

[31] W. Yang, M. R. Prasad, and T. Xie, "A Grey-box Approach for Automated GUI-model Generation of Mobile Applications," in *FASE*, 2013.

[32] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?" in *FSE*, 2016.

[33] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case," in *ICSE-SEIP*, 2017.

[34] K. Inkumsah and T. Xie, "Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution," in *ASE*, 2008.

[35] Google, "Android App Components," 2018. [Online]. Available: https://developer.android.com/guide/components/fundamentals#Components

[36] Google, "Android Platform Architecture," 2018. [Online]. Available: https://developer.android.com/guide/platform/

[37] Google, "ART and Dalvik," 2017. [Online]. Available: https://source.android.com/devices/tech/dalvik/

[38] Google, "Android Dalvik Executable format," 2017. [Online]. Available: https://source.android.com/devices/tech/dalvik/dex-format

[39] Google, "Android 64K Method limit," 2018. [Online]. Available: https://developer.android.com/studio/build/multidex

[40] B. Insider, "WeChat has hit 1 billion monthly active users," 2018. [Online]. Available: http://www.businessinsider.com/wechat-has-hit-1-billion-monthly-active-users-2018-3

[41] K. Mao, M. Harman, and Y. Jia, "Crowd Intelligence Enhances Automated Mobile Testing," in *ASE*, 2017.

[42] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, 2002.

[43] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, "An Introduction to MCMC for Machine Learning," *Machine Learning*, vol. 50, no. 1, Jan 2003.

[44] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A lightweight test input generator for Android," 2018. [Online]. Available: https://github.com/honeynet/droidbot

[45] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, 1987.

[46] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *PLDI*, 2005.

[47] S. Anand, "ELLA: A Tool for Binary Instrumentation of Android Apps," 2016. [Online]. Available: https://github.com/saswatanand/ella

[48] Google, "Logcat command-line tool," 2021. [Online]. Available: https://developer.android.com/studio/command-line/logcat

[49] Google, "Android Debug Bridge (ADB)," 2021. [Online]. Available: https://developer.android.com/studio/command-line/adb

[50] X. He, "Python wrapper of Android uiautomator test tool," 2018. [Online]. Available: https://github.com/xiaocong/uiautomator

[51] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," 1999.

[52] "TOLLER Artifacts," 2021. [Online]. Available: https://github.com/TOLLER-Android/main

[53] "Espresso Test Recorder," 2021. [Online]. Available: https://developer.android.com/studio/test/espresso-test-recorder.html

[54] JesusFreke, "smali/baksmali," 2021. [Online]. Available: https://github.com/JesusFreke/smali

[55] T. Gu, "MiniTrace," 2020. [Online]. Available: http://gutianxiao.com/ape/

[56] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An Empirical Analysis of UI-based Flaky Tests," in *ICSE*, 2021.

[57] F. P. Brooks, *The Mythical Man-Month (Anniversary Ed.)*, 1995.

[58] VET Artifacts, 2021. [Online]. Available: https://github.com/VET-UI-Testing/main

[59] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu, and J. Lu, "DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps," in *CCS*, 2019.

[60] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao, "IconIntent: Automatic Identification of Sensitive UI Widgets Based on Icon Classification for Android Apps," in *ICSE*, 2019.

[61] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning Design Semantics for Mobile Apps," in *UIST*, 2018.

[62] Š. Packevičius, D. Barisas, A. Ušaniov, E. Guogis, and E. Bareiša, "Text Semantics and Layout Defects Detection in Android Apps Using Dynamic Execution and Screenshot Analysis," in *ICIST*, 2018.

[63] Google, "Android View," 2021. [Online]. Available: https://developer.android.com/reference/android/view/View

[64] Google, "Android ViewGroup," 2021. [Online]. Available: https://developer.android.com/reference/android/view/ViewGroup

[65] Y.-M. Baek and D.-H. Bae, "Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria," in *ASE*, 2016.

[66] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications," in *ICSME*, 2017.

[67] K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," *SIAM Journal on Computing*, 12 1989.

[68] L. A. Wolsey and G. L. Nemhauser, *Integer and combinatorial optimization*, 1999.

[69] A. Schrijver, *Theory of Linear and Integer Programming*, 1986.

[70] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-Travel Testing of Android Apps," in *ICSE*, 2020.

[71] A. W. Services, "AWS Device Farm," 2022. [Online]. Available: https://aws.amazon.com/device-farm/

[72] S. Labs, "Sauce Labs," 2022. [Online]. Available: https://saucelabs.com/

[73] Perfecto, "Perfecto: Web & Mobile App Testing — Continuous Testing," 2022. [Online]. Available: https://www.perfecto.io/

[74] LambdaTest, "LambdaTest: Cross Browser Testing Cloud," 2022. [Online]. Available: https://www.lambdatest.com/

[75] Kobiton, "Kobiton: Mobile Device Testing," 2022. [Online]. Available: https://kobiton.com/

[76] D. Garbar, "How Often Should You Update Your Mobile App?" 2018. [Online]. Available: https://www.apptentive.com/blog/2018/12/27/how-often-should-you-update-your-mobile-app/

[77] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu, "Software protection on the go: A large-scale empirical study on mobile app obfuscation," in *ICSE*. IEEE, 2018.

[78] T. Cai, Z. Zhang, and P. Yang, "Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd." in *AST*, 2020.

[79] J. Sun, T. Su, J. Li, Z. Dong, G. Pu, T. Xie, and Z. Su, "Understanding and Finding System Setting-Related Defects in Android Apps," in *ISSTA*, 2021.

[80] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang, and H. Duan, "TextExerciser: Feedback-driven Text Input Exercising for Android Applications," in *S&P*, 2020.

[81] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully Automated Functional Fuzzing of Android Apps for Detecting Non-Crashing Logic Bugs," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, oct 2021.

[82] "appetizer-toolkit," 2017. [Online]. Available: https://github.com/appetizerio/appetizer-toolkit

[83] Z. Qin, Y. Tang, E. Novak, and Q. Li, "MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications," in *ICSE*, 2016.

[84] "Ranorex," 2021. [Online]. Available: http://www.ranorex.com/mobile-automation-testing.html

[85] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and Touch-Sensitive Record and Replay for Android," in *ICSE*, 2013.

[86] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet Lightweight Record-and-Replay for Android," in *OOPSLA*, 2015.

[87] "Culebra," 2021. [Online]. Available: https://github.com/dtmilano/AndroidViewClient/wiki/culebra

[88] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating Video Recordings of Mobile App Usages into Replayable Scenarios," in *ICSE*, 2020.

[89] J. Qian, Z. Shang, S. Yan, Y. Wang, and L. Chen, "RoScript: A Visual Script Driven Truly Non-Intrusive Robotic Testing System for Touch Screen Applications," in *ICSE*, 2020.

[90] "Bot-bot," 2021. [Online]. Available: http://imaginea.github.io/bot-bot/index.html

[91] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: Cross-Platform User-Interaction Record and Replay for the Fragmented Android Ecosystem," in *ISPASS*, 2015.

[92] "Robotium: A unit test framework for Android." [Online]. Available: https://github.com/robotiumtech/robotium

[93] J. Guo, S. Li, J.-G. Lou, Z. Yang, and T. Liu, "Sara: Self-Replay Augmented Record and Replay for Android in Industrial Cases," in *ISSTA*, 2019.

[94] O. Sahin, A. Aliyeva, H. Mathavan, A. Coskun, and M. Egele, "Towards Practical Record and Replay for Mobile Applications," in *DAC*, 2019.

[95] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor," in *EuroSys*, 2014.

[96] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI Testing for Android Applications," in *ASE*, 2017.

[97] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI Screenshots for Search and Automation," in *UIST*, 2009.

[98] E. Alegroth, M. Nass, and H. H. Olsson, "JAutomate: A Tool for System- and Acceptance-test Automation," in *ICST*, 2013.

[99] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *ICCV*, 2017.

[100] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated Identification of Cross-Browser Issues in Web Applications," in *ICSM*, 2010.

[101] M. He, G. Wu, H. Tang, W. Chen, J. Wei, H. Zhong, and T. Huang, "X-Check: A Novel Cross-Browser Testing Service Based on Record/Replay," in *ICWS*, 2016.

[102] Z. Xu and J. Miller, "Cross-Browser Differences Detection Based on an Empirical Metric For Web Page Visual Similarity," *TIT*, 2018.

[103] RaiMan, "SikuliX by RaiMan," 2021. [Online]. Available: http://sikulix.com

[104] T. D. White, G. Fraser, and G. J. Brown, "Improving Random GUI Testing with Image-Based Widget Detection," in *ISSTA*, 2019.

[105] K. B. Dhanapal, K. S. Deepak, S. Sharma, S. P. Joglekar, A. Narang, A. Vashistha, P. Salunkhe, H. G. Rai, A. A. Somasundara, and S. Paul, "An Innovative System for Remote and Automated Testing of Mobile Phone Applications," in *SRII*, 2012.

[106] K. Mao, M. Harman, and Y. Jia, "Robotic Testing of Mobile Apps for Truly Black-Box Automation," *IEEE Software*, 2017.

[107] J. Qian, Z. Shang, S. Yan, Y. Wang, and L. Chen, "RoScript: A Visual Script Driven Truly Non-Intrusive Robotic Testing System for Touch Screen Applications," in *ICSE*, 2020.

[108] R. Wernersson, "Robot Control and Computer Vision for Automated Test System on Touch Display Products," M.S. thesis, 2015.

[109] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust Log-Based Anomaly Detection on Unstable Log Data," in *ESEC/FSE*, 2019.

[110] Z. Zhao, S. Cerf, R. Birke, B. Robu, S. Bouchenak, S. Ben Mokhtar, and L. Y. Chen, "Robust Anomaly Detection on Unreliable Data," in *DSN*, 2019.

[111] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *CCS*, 2017.

[112] C. Monni and M. Pezzè, "Energy-Based Anomaly Detection a New Perspective for Predicting Software Failures," in *ICSE-NIER*, 2019.

[113] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan, "The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure," in *SOSP*, 2019.

[114] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting Developers of Big Data Analytics Applications When Deploying on Hadoop Clouds," in *ICSE*, 2013.

[115] A. R. Chen, "An Empirical Study on Leveraging Logs for Debugging Production Failures," in *ICSE-Companion*, 2019.

[116] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "REPT: Reverse Debugging of Failures in Deployed Software," in *OSDI*, 2018.

[117] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs," in *ESEC/FSE*, 2019.

[118] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach," in *SOSP*, 2017.

[119] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "Lprof: A Non-Intrusive Request Flow Profiler for Distributed Systems," in *OSDI*, 2014.

[120] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle," in *OSDI*, 2016.

[121] P. R. Mateo and M. P. Usaola, "Parallel mutation testing," *Software Testing, Verification and Reliability*, 2013.

[122] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in Parallel," in *ISSTA*, 2007.

[123] M. Staats and C. Păsăreanu, "Parallel Symbolic Execution for Structural Test Generation," in *ISSTA*, 2010.

[124] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel Symbolic Execution for Automated Real-World Software Testing," in *EuroSys*, 2011.

[125] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic Text Input Generation for Mobile Testing," in *ICSE*, 2017.

[126] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[127] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[128] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *ICSE*, 2012.

[129] P. Zhang and S. Elbaum, "Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, sep 2014.

[130] M. Pan, Y. Lu, Y. Pei, T. Zhang, and X. Li, "Preference-Wise Testing of Android Apps via Test Amplification," *ACM Transactions on Software Engineering and Methodology*, jan 2022.

[131] J. Wang, X. Bai, H. Ma, L. Li, and Z. Ji, "Cloud API Testing," in *ICSTW*, 2017.

[132] J. Wang, X. Bai, L. Li, Z. Ji, and H. Ma, "A Model-Based Framework for Cloud API Testing," in *COMPSAC*, vol. 2, 2017.

[133] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *ICSE*, 2019.

[134] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential Regression Testing for REST APIs," in *ISSTA*, 2020.