Navigating Mobile Testing Evaluation: A Comprehensive Statistical Analysis of Android GUI Testing Metrics

Yuanhong Lan
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
yhlan@smail.nju.edu.cn

Yifei Lu*
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
lyf@nju.edu.cn

Minxue Pan*
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
mxp@nju.edu.cn

Xuandong Li
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
lxd@nju.edu.cn

ABSTRACT

The prominent role of mobile apps in daily life has underscored the need for robust quality assurance, leading to the development of various automated Android Graphical User Interface (GUI) testing approaches. Code coverage and fault detection are two primary metrics for evaluating the effectiveness of these testing approaches. However, conducting a reliable and robust evaluation based on the two metrics remains challenging, due to the imperfections of the current evaluation system, with a tangle of numerous metric granularities and the interference of multiple nondeterminism in tests. For instance, the evaluation solely based on the mean or total numbers of detected faults lacks statistical robustness, resulting in numerous conflicting conclusions that impede the comprehensive understanding of stakeholders involved in Android testing, thereby hindering the advancement of Android testing methodologies. To mitigate such issues, this paper presents the first comprehensive statistical study of existing Android GUI testing metrics, involving extensive experiments with 8 state-of-the-art testing approaches on 42 diverse apps, examining aspects including statistical significance, correlation, and variation. Our study focuses on two primary areas: ① The statistical significance and correlation between test metrics and among different metric granularities. ② The influence of test randomness and test convergence on evaluation results of test metrics. By employing statistical analysis to account for the considerable influence of randomness, we achieve notable findings: ① Instruction, Executable Lines Of Code (ELOC), and method coverage demonstrate notable consistency across both significance evaluation and mean value evaluation, whereas the evaluation on Fatal Errors compared to Core Vitals, as well as all errors versus the well-selected errors, reveals a similarly high level of consistency. ② There are evident inconsistencies in the code coverage and fault detection results, indicating both two metrics should be considered for comprehensive evaluation. ③ Code coverage typically exhibits greater stability and robustness in evaluation compared to fault detection, whereas fault detection is quite unstable even with the maximum test rounds ever used in previous research studies. ④ A moderate test duration is sufficient for most approaches to showcase their comprehensive overall effectiveness on most apps in both code coverage and fault detection, indicating the possibility of adopting a moderate test duration to draw preliminary conclusions in Android testing development. These findings inform practical recommendations and support our proposal of an effective framework to enhance future mobile testing evaluations.

*Corresponding authors.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695476

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **General and reference** → **Metrics**; **Evaluation**; • **Mathematics of computing** → **Probability and statistics**; • **Human-centered computing** → *Graphical user interfaces.*

KEYWORDS

Mobile testing, Testing metrics and evaluation, Statistical analysis

ACM Reference Format:
Yuanhong Lan, Yifei Lu, Minxue Pan, and Xuandong Li. 2024. Navigating Mobile Testing Evaluation: A Comprehensive Statistical Analysis of Android GUI Testing Metrics. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695476

1 INTRODUCTION

With the increasing influence of mobile applications (hereafter called apps) on individuals' daily routines, ensuring mobile app quality has become critically important. Various automated Graphical User Interface (GUI) testing approaches (e.g., [16, 20, 25, 26, 33, 35, 40]) have been proposed to improve the test effectiveness as well as alleviate the manual test effort for mobile apps.

In the context of these research efforts, it is a common practice to evaluate the testing effectiveness of various approaches via two important metrics: code coverage and fault detection. These metrics are essential and the most frequently used ones in mobile testing. Nevertheless, despite the adoption of these metrics, experimental outcomes across mobile testing research studies can still yield different, or even contradictory conclusions, as we have observed.

Take app *MyExpenses*, a widely-used open-source app with 50,000 Executable Lines Of Code (ELOC) and millions of users, as an example. In terms of fault detection in total, Stoat [36], a guided, stochastic Model-Based GUI testing approach, demonstrated its superiority over the classical fuzzing testing approach Monkey [16],

as evidenced by Su et al.'s experiments [36]. Subsequently, experiments conducted by Gu et al.[20], however, arrived at a completely opposite conclusion. Further complicating the matter, the experimental findings of Pan et al. [33], which aligned with those of Su et al., added to the confusion surrounding these results. In the context of code coverage, similar observations can be noted between the experiments conducted by Wang et al. [41] and Ran et al. [34]. Across a total of 15 apps, including the globally recognized apps *Duolingo* and *Chrome*, Monkey significantly outperformed Stoat in mean Activity coverage in the Wang et al.'s experiments, whereas completely opposites were observed in Ran et al.'s experiments.

These previously unexplained inconsistencies in the experimental results necessitate a reconsideration of their origins. However, it remains unclear whether these inconsistencies arise from variations in metric configurations or from the inherent instability of the metrics themselves, although we did observe some diverse usages of these two metrics among these testing approaches. Furthermore, we question whether the common practice of using the mean and total numbers, which are the main and often the only methods applied for code coverage and fault detection statistics among most testing approaches [7, 10, 20, 25, 26, 33, 35–37, 40–42], for testing effectiveness comparison may contribute to conflicting experimental conclusions. Although several studies [4, 7, 37, 41–43] have focused on the testing effectiveness of different mobile testing approaches, unfortunately, none could ever address these critical questions.

To bridge this gap, this paper proposes the first empirical study on these two widely recognized Android GUI testing metrics, with the purpose of not only demystifying the objective laws and characteristics of these metrics and their usages but also offering actionable insights to developers, users, and critics of mobile testing approaches. Moreover, another novel aspect of this study is the in-depth application of statistic analysis, particularly a hypothesis testing method, namely the *Independent t-Test* [39], to assess the statistical significance between different testing approaches with respect to code coverage and fault detection. With the *Independent t-Test*, this paper delivers a robust, comprehensive, and reliable data analysis on test effectiveness, which accounts for the variability, controls the error rates, and therefore mitigates the potential bias in experimental results. In addition, other advanced statistical methods, such as *Pearson Correlation Coefficient* [1] and *Coefficient of Variation* [5], are employed in this paper to investigate the correlation and the randomness of the experimental results. Thus, unlike studies that merely compare mean or total values, this paper provides more reliable and rigorous conclusions.

Specifically, we focus on the following four research questions:

- **RQ1: Granularity of Test Metrics.** Code coverage and fault detection are frequently utilized metrics, yet their granularity can vary vastly in the detailed statistics across different research. For instance, units of code coverage statistics can include instructions [25, 33, 35, 40], ELOC [4, 7, 10, 36], methods [10, 20, 41], and even Activities [26, 35, 41, 43]. Given the variability in metric granularity, it is imperative to pose the following inquiries: how do the granularities of these metrics influence the effectiveness of testing approach evaluations? Do code coverage and fault detection at different granularities exhibit consistent significance, or do they produce conflicting observations?
- **RQ2: Relations Across Test Metrics.** In most cases, both code coverage and fault detection are used for test effectiveness evaluation, and in general speaking, the more effective a testing approach is, the more code coverage and faults it can achieve or reveal. However, the correlation between the code coverage and the fault detection capabilities of testing approaches has not been thoroughly studied. Maybe they are interchangeable? Thus, it is interesting to explore to what extent code coverage results correlate with fault detection in testing approach comparisons. Do testing approaches exhibit consistent performance in both code coverage and fault detection with the same test?
- **RQ3: Test Randomness on Test Metrics.** Given the stochastic nature of Android testing, experimental results can exhibit considerable variability. Previous research has designed their experiments with repeated trials to reduce potential biases, with the number of test rounds varying from three [26, 40, 41] to ten [4, 7, 35]. The choice of this setting raises several relevant questions: how does randomness affect the evaluation in terms of code coverage and fault detection? To what extent do repeated experiments with various test rounds mitigate this bias?
- **RQ4: Test Convergence on Test Metrics.** Test time, another important experimental design in Android testing, has been set to analyze whether testing approaches can maintain high effectiveness within constrained timeframes. Like test rounds, test time has diverse settings in previous research ranging from 1 hour [4, 7, 33, 35] to 18 hours [42], each of which has claimed the rationality of their settings. Therefore, here raises the following questions: how do different testing approaches demonstrate their effectiveness in terms of test convergence on code coverage and fault detection? Do testing approaches manifest consistency in effectiveness under different time points?

Experiments were conducted with 3-hour tests across 8 State-Of-The-Art (SOTA) testing approaches on 42 diverse open-source apps, each repeated 10 times. Here are our main findings: ① The performances of the three fine-grained code coverage granularities (instruction, ELOC, and method) in both significance and mean value evaluations are nearly consistent. Therefore, when achieving finer-grained coverages poses challenges, method coverage can serve as the representative granularity. The evaluation results from Fatal Errors [14] align closely with those from Core Vitals [13], and the results from broad error types are highly consistent with those from well-selected types. Hence, it is recommended to include both groups for fault detection evaluation. ② Despite a moderate positive correlation, notable inconsistencies are observed between code coverage and fault detection when used for approach comparison evaluations. This suggests that applying both metrics simultaneously is imperative for comprehensive evaluations. ③ The evaluation results from code coverage are evidently more stable compared to those from fault detection. Even with the maximum test round settings among the existing studies, substantially increasing variations are observed in fault detection results. This highlights the need for particular caution when drawing conclusions based solely on mean values of fault detection results. ④ With the evident convergence trend in both metrics and the high correlation among

Table 1: Overview of Previous Empirical Studies and Representative Testing Approaches in Android GUI Testing

	Study / Approach	Venue	Category	Settings		Code Coverage				Fault Detection		Other Metrics
				Time	Round	INST	ELOC	METH	ACTV	Logcat	NewIssue	
Empirical Studies	Su et al. [37]	ESEC/FSE'21		6h	5							BugBenchmark
	Behrang et al. [4]	ASE'20		1h	10		✓					
	Wang et al. [41]	ASE'18		3h	3			✓	✓	E		EaseOfUse
	Zheng et al. [43]	ICSE'17-SEIP		12h			✓		✓			
	Zeng et al. [42]	FSE'16-Ind		18h	5		✓		✓			
	Choudhary et al. [7]	ASE'15		1h	10		✓			E		EaseOfUse, Compatibility
Testing Approaches	DQT [25]	ICSE'24	Deep-RL-Based	2h	5	✓				F	✓	
	ARES [35]	TOSEM'22	Deep-RL-Based	1h	10	✓			✓	F		
	ComboDroid [40]	ICSE'20	Systematic	12h	3	✓				E	✓	
	Q-testing [33]	ISSTA'20	Tabular-RL-Based	1h	4	✓				E+	✓	
	TimeMachine [10]	ICSE'20	Systematic	6h	5		✓	✓		F		
	APE [20]	ICSE'19	Model-Based	1h	5	✓		✓	✓	F	✓	
	Humanoid [26]	ASE'19	Supervised-Learning	1h	3		✓		✓			
	Stoat [36]	ESEC/FSE'17	Model-Based	3h	5		✓			E+	✓	
	Monkey [16]	Google	Random-Based									

results at different time points, a moderate test duration is generally sufficient to draw preliminary conclusions about the overall effectiveness in the majority of cases.

These findings reveal deficiencies in current practices and shed light on the application of code coverage and fault detection as well as the related crucial settings for test effectiveness evaluation. With practical guidance to conduct more reliable and robust evaluations of testing effectiveness, this study seeks to enhance the overall reliability and validity of testing evaluation practices in the field.

In addition, we have developed and released an effective automated evaluation framework based on the statistical analysis techniques employed in this study. This framework aims to provide an off-the-shelf statistical evaluation in Android GUI testing and serve as a reference of effectiveness evaluation for other fields.

In summary, we have the following main contributions:

- A survey of the main existing studies and testing techniques in Android GUI testing, identifying evaluation challenges.
- An extensive experiment involving 8 SOTA testing approaches on 42 diverse well-recognized open-source apps with 10 repetitons.
- A comprehensive statistical analysis of the existing primary Android GUI testing metrics, achieving notable findings, practical suggestions, and an effective automated evaluation framework.
- Our study data and framework are publicly available at https://github.com/Yuanhong-Lan/AndroTest24 for future study.

2 STUDY SETUPS

Table 1 presents an overview of previous research on Android GUI testing, divided into two sections: 6 previous empirical studies (top half) and 9 representative testing approaches from recent years (bottom half), on which we set up our study. Each entry details the experimental settings, including test time and test round, as well as the metrics and granularities utilized.

Given the dynamic and evolving nature of Android GUI testing, with a continuous emergence of new methodologies, a systematic method was employed for selecting representative ones. First, we collected research studies from the past 10 years on Android GUI testing from leading conferences and journals in Software Engineering, resulting in 68 papers. With snowballing, the number of candidates expands to 131. After excluding empirical studies and approaches with particular testing purposes, 24 approaches remained. Next, following previous studies, these approaches were classified into four categories: Random-Based, Model-Based, Systematic, and Machine-Learning-Based, where Machine-Learning-Based testing was further divided into Supervised-Learning-Based, Tabular-RL-Based, and Deep-RL-Based, due to its recent popularity. We selected approaches frequently included for comparison evaluations, with a threshold of 5 times. Besides, to ensure diversity, we included the latest two approaches if none met the threshold in a specific category. Eventually, 9 representative testing approaches listed in Table 1 were selected, showcasing various representative methodologies.

2.1 Testing Approaches

Substantial effort was invested in configuring all 9 testing approaches in Table 1. However, TimeMachine [10] had to be excluded due to its requirement for modifying and substituting system files, which disrupted the uniformity of the testing environment and affected the functionality of other approaches by altered system dependencies. Consequently, our study included 8 testing approaches: Monkey [16], Stoat [36], APE [20], ComboDroid [40], Humanoid [26], Q-testing [33], ARES [35], and DQT [25].

To ensure the validity of our experiments, we adopted the default settings provided in the respective paper or source code repository of each testing approach. The default settings, recommended by their developers, reduce interference with the proposed performance of each approach and are generally considered best practices. Monkey [16] stands as a well-established Random-Based baseline. Following previous studies [7, 20, 25, 33], we set a delay of 200 milliseconds between events for Monkey. Stoat [36] and APE [20] exemplify prominent Model-Based testing methodologies. Stoat adheres closely to the traditional Model-Based paradigm, while APE follows a more flexible dynamic paradigm. ComboDroid [40] emerges as a cutting-edge systematic testing approach. The remaining four approaches are situated at the forefront of Machine-Learning-Based testing. Humanoid [26] employs supervised learning, while Q-testing [33] is grounded in Tabular Reinforcement Learning (RL). ARES [35] and DQT [25] represent recent advances in Deep-RL-Based testing. For ARES, we selected

the best-performing SAC algorithm among its Deep RL algorithms and consulted with its authors due to its numerous settings.

2.2 App Benchmark

We propose a comprehensive app benchmark, AndroTest24, which builds upon the well-known AndroTest app benchmark [7] and incorporates contributions from **all the open-source app benchmarks adopted by the research studies in Table 1**. We focus on open-source apps due to their widespread adoption in previous studies and the necessity of instrumentation for precise fine-grained app information, such as ELOC coverage and instruction coverage, which mandates access to app source code. By merging all these open-source benchmarks, we achieved 132 candidate apps. We meticulously filter out outdated apps, i.e., those that have not received any commits within the past three years, to ensure the relevance and realism of our research. Eventually, we identified 42 active apps for inclusion and adopted the latest app version.

Table 2: Overview of Our App Benchmark AndroTest24

	Basic Information					App Scale			
ID	App Name	Star	GPD	Version	Category	ACTV	METH	ELOC	INST
1	Signal⊗	24.7k	10m+	6.3.3	Communication	84	44,150	174,574	904,470
2	WordPress⊗	2.9k	10m+	23.5rc2	Productivity	119	31,701	131,725	742,156
3	CoronaWarn⊗	2.4k		2.28.3	Health	12	17,088	70,154	459,709
4	Tachiyomi⊗	21.2k		0.13.6	Entertainment	13	9,581	45,341	412,967
5	DuckDuckGo⊗	3.3k	50m+	5.140.0	Browser	55	16,673	60,207	363,204
6	K9Mail⊗	9.2k	5m+	6.712	Communication	33	13,134	56,831	349,645
7	Firefox⊗	6.5k	100m+	110.0a1	Browser	17	11,526	53,428	348,086
8	Wikipedia⊗	2.2k	50m+	2.7.50453	Reference	52	12,192	45,754	334,262
9	MyExpenses	681	1m+	3.4.5	Finance	44	9,362	50,658	300,635
10	AnkiDroid	7.7k	10m+	2.16.2	Education	35	9,233	43,663	259,600
11	SuntimesWidget⊗	307		0.15.6	Daily	32	8,278	46,453	232,988
12	Conversations	4.2k	100k+	2.10.10	Communication	36	6,596	42,740	191,099
13	NewPipe⊗	27.9k		0.23.3	News	14	6,295	34,710	149,421
14	AmazeFileManager	5k	1m+	3.8.5	System Tools	10	4,936	31,680	144,944
15	MoneyManagerEx	423		2021.05.13	Finance	50	5,085	28,712	127,964
16	BookCatalogue	376	500k+	5.3.0-5	Productivity	35	3,966	24,663	117,778
17	AntennaPod	5.6k	500k+	3.1.0-beta2	Audio	10	5,114	27,669	116,750
18	LBRY⊗	2.5k	500k+	0.17.1	Communication	6	3,482	16,460	89,679
19	ConnectBot	2.3k	5m+	1.9.9	System Tools	11	1,200	8,977	70,584
20	RunnerUp	703	50k+	2.6.0.2	Fitness	16	2,519	15,969	70,184
21	Timber	7k	100k+	1.8	Music	9	2,262	12,111	54,691
22	APhotoManager	222		0.8.3.200315	Photography	11	2,055	11,451	54,313
23	BetterBatteryStats⊗	598	100k+	3.4.0	System Tools	12	1,324	11,189	52,918
24	Vanilla	1.1k	1m+	1.3.1	Music	13	1,437	10,482	47,367
25	AnyMemo	151	100k+	10.11.7	Education	28	1,997	10,042	45,914
26	SimpleTask	545		11.0.1	Lifestyle	14	1,666	8,145	45,361
27	LoopHabitTracker	7.2k	5m+	2.2.0	Productivity	11	1,768	9,097	44,896
28	TranslateYou	742		7.1	Reference	3	1,113	3,838	43,875
29	KeePassDroid	1.4k	1m+	2.6.8	Privacy	15	1,811	9,598	38,496
30	Materialistic	2.3k		3.3	News	23	1,929	7,782	33,967
31	AlarmClock	462	1m+	3.15.02	Daily	5	1,359	4,256	29,356
32	Currencies⊗	189	1k+	1.20.4	Finance	3	589	2,600	18,879
33	Notes	90	10k+	1.4.2	Lifestyle	12	705	3,511	18,667
34	PasswordManager	20	50+	1.1	Privacy	4	538	2,869	15,477
35	Swiftp	710	1k+	3.1	System Tools	4	447	3,690	14,408
36	Aard2	405	10k+	0.54	Reference	2	512	2,648	11,290
37	Diary⊗	261		1.102	Lifestyle	4	269	2,345	9,333
38	ArxivExplorer	56	10k+	4.1.1	Education	5	446	1,859	8,923
39	SimpleDraw	499	100k+	6.9.6	Art	8	297	1,503	8,096
40	KindMind			1.2.1_BETA	Health	5	234	1,689	7,652
41	CEToolbox	5	500+	1.5.0	Business	7	156	1,670	6,473
42	WhoHasMyStuff		5k+	1.1.0	Productivity	7	135	759	4,533

Table 2 provides an overview of AndroTest24, listing apps in descending order based on the number of instructions. The table is further divided into two parts. The left part includes basic information about the apps, including the name, GitHub stars (if available), Google Play Downloads (GPD, if available), the version, and the category. Whereas the right part describes the scales of the apps, detailing the number of activities (ACTV), methods (METH), ELOC, and instructions (INST). It is important to note that 13 apps were unable to undergo the static analysis conducted by ComboDroid [40] and resulted in crashes; these apps are marked with ⊗ in the table.

Notably, although all these 42 apps are open-source, they exhibit a wide range of scales, varying from 2 to 119 activities, 135 to 44k methods, 759 to 174k ELOC, and 4k to 904k instructions. These apps span 19 different categories, underscoring the diversity of our AndroTest24. Additionally, many of these apps also function as real-world commercial apps available on various app stores, with several achieving significant global traction. For instance, over 70% (31 out of 42) of these apps are available on Google Play, the world's largest Android app store, with apps like *Signal* and *Firefox* boasting over 100 million installations each.

2.3 Experimental Configurations

To ensure the reliability and fairness of our tests across all the testing approaches on various apps, we meticulously configured our experimental environment at all levels.

Hardward Configurations. Experiments were conducted on the same physical machine with Android emulators [15] for stable, replicable environments across tests. We selected Android 9.0 for its compatibility across existing testing approaches and supportive advantages [19]. Each emulator was configured with 4GB RAM to ensure smooth operation for large apps like *Signal* and *WordPress*.

Testing Configurations. All emulators originated from the same parent to maintain consistency. Each App was assigned to a specific emulator to minimize emulator variability. We used snapshots to ensure a uniform start state for each test. For apps that require a log-in state for normal operation (e.g., *Signal*, *WordPress*), a logged-in status with a test account is provided within the snapshot.

Experimental Settings. The experimental time was set to 3 hours, aligning with the testing timeframes of most prior studies [4, 7, 20, 25, 26, 33, 35, 36, 41]. For Stoat [36], we follow previous practices [25, 33] and allocate half time for modeling and half for sampling. In addition, we adopted the highest number of test rounds used among the studies listed in Table 1, i.e., 10 rounds. Consequently, we performed a total of 42 Apps (ComboDroid [40] supports only 29 apps) × 8 Approaches × 10 Runs = 3230 valid Tests. Given the complexity of the testing process, including data collection and analysis, any interruption could result in invalid test outcomes. The entire experimental process required approximately 3230 Tests × 3 Hours × (1 + 0.3 for Coverage Calculations + 0.5 for Retries) ≈ 727 CPU Days. Even with four emulators running concurrently, this process lasted over six months.

Experimental Data Collection. To achieve precise fine-grained code coverage, we instrumented the source code of apps and employed Jacoco [11], with coverage files collected every 10 seconds and calculated after each test. For fault detection, we followed the common practice [7, 10, 20, 25, 33, 35, 36, 40, 41], monitoring Logcat [18] at runtime and extracting exceptions with the following steps: ① Filter by package information to exclude unrelated ones. ② Deduplicate those with the same exception type and source code reference. ③ Categorization (refer to Section 2.4). Given the complexity of the process, for better reliability, three authors independently extract faults, followed by mutual proofreading, resulting in a total time investment of approximately three Man-Months.

2.4 Metric Granularity

As depicted in Table 1, we are incorporating widely recognized and quantifiable evaluation metrics from previous research, resulting in two primary metrics, each with four levels of granularities.

Code Coverage. The selection of granularities for code coverage is straightforward. Prior works predominantly focus on four levels: instruction coverage (INST), ELOC coverage, method coverage (METH), and Activity coverage (ACTV), as illustrated in Table 1. Accordingly, we incorporated these four for code coverage.

Fault Detection. As indicated in Table 1, previous research predominantly assesses the fault detection ability from two specific perspectives: filtering exceptions from Logcat [18] or reporting new issues to developers. While the ultimate goal of fault detection is to identify and report faults to developers for resolution, quantifying this process for comparative, statistical analysis poses challenges. Therefore, in this paper, we focus solely on extracting exceptions from Logcat, which is also the base of fault fixing.

Table 3: Granularity of Fault Detection

ID	Description	Source	Logcat Filter
F	Fatal Errors	[10, 20, 25, 35]	*:S *:F AndroidRuntime:E
V	Core Vitals	[13]	*:S *:F AndroidRuntime:E ActivityManager:E
E+	Selected Errors	[33, 36]	*:S *:F AndroidRuntime:E ActivityManager:E SQLiteDatabase:E WindowManager:E ActivityThread:E Parcel:E
E	Errors	[7, 40, 41]	*:S *:F *:E

Table 3 enumerates the four levels of fault detection granularities, detailing their identifiers, descriptions, sources, and the Logcat filters necessary for their extraction. From the bottom to the top of the table, the scope of concern becomes more specific, while the associated faults typically become more severe. At the top, Level F, widely adopted by prior works [10, 20, 25, 35], correlates strongly with Fatal Errors [14], which frequently result in app crashes and severely impact user experiences. Level V, introduced as Core Vitals [13] by Google, encompasses not only Fatal Errors but also Application Not Responding (ANR) Errors [17], another critical user experience disaster. Subsequently, Level E+, utilized by [33, 36], incorporates additional error types, such as window leaks and database errors, which concern developers but may not necessarily lead to fatal consequences. Additionally, some studies [7, 40, 41] extract errors from Logcat without explicitly specifying type constraints in their papers or available source codes. In such cases, we categorize them under Level E, covering all potential app-related errors presented in Logcat.

2.5 Statistical Analysis

The nature of randomness in Android testing originates from two primary sources: ① The high complexity of modern Android apps results in a multitude of uncertainties, influenced by diverse testing environments, network conditions, and contexts subscribed from remote servers, etc. ② Existing automated testing approaches—no matter based on models, RL algorithms, or systematic methods—often incorporate some random strategies to enhance their exploration ability, thereby promoting test effectiveness.

When it comes to effectiveness evaluation, randomness poses a significant challenge, garnering increasing attention in recent years [25, 37]. The pervasive nature of randomness raises the question of whether a test result accurately reflects the effectiveness of the testing tool or is merely a consequence of random variation.

To alleviate the challenge of randomness, while previous efforts have focused on comparing mean or total values through repeated experiments, we first approach it from a more statistically rigorous perspective, where three main statistical methods are employed.

Significance Analysis. We employ an *Independent t-Test* [39] to determine whether there is a statistically significant difference between the means of two sets of testing results while taking their variance or distribution into account. From the granularity perspective, each test result—i.e., how a tool performs at a metric granularity during a single test run on an app—is considered as a sample. To illustrate this analysis, consider a simple example: Tool A conducts 10 tests on App x, with its performance on metric granularity γ denoted as $(a_1, a_2, ..., a_{10})$. Similarly, Tool B conducts 10 tests on App x, and its performance on γ is represented as $(b_1, b_2, ..., b_{10})$. By t-test, we can compute the significance value $p_{\gamma xAB}$ of the difference between the means of Tool A and Tool B under App x and metric granularity γ. This significance value is then evaluated against the predefined significance level (commonly $\alpha = 0.05$). With this setup, we obtain a total of 42 Apps (29 for ComboDroid [40]) $\times C_8^2$ Pairs (pairwise comparison of approaches) = 1085 groups of comparison results for each metric granularity.

The two metrics, i.e., code coverage and fault detection, have their respective four granularities. To compare Tool A and Tool B on App x **at the metric level**, we calculate the following dimensions to *assess different granularities within this metric.*

- S_{gra_sig}**: Granularity Significance.** A comparison result is deemed statistically significant if the p-value $p <= \alpha$ ($\alpha = 0.05$).
- S_{sig_cmp}**: Significant Comparison.** If *more than two-thirds (absolute majority) of the granularities within a metric* yield statistically significant comparison results and in the same direction (all indicating that Tool A > Tool B, or vice versa), significant comparison result is observed *at the metric level.*
- S_{cst_cmp}**: Consistent Comparison.** Like S_{sig_cmp} but requiring the comparison results from *all granularities within a metric* to be statistically significant and in the same direction.
- S_{non_sig}**: Non-significant with** S_{sig_cmp}**.** With S_{sig_cmp} in a metric, the comparison result from a certain granularity within the metric is not statistically significant.
- $S_{conflict}$**: Conflict with** S_{sig_cmp}**.** With S_{sig_cmp} in a metric, the comparison result from a certain granularity within the metric is statistically significant but indicates an *opposite direction* (e.g., while S_{sig_cmp} shows Tool A > Tool B, the current granularity indicates Tool A < Tool B).

In this evaluation: ① S_{gra_sig} delineates acceptable granularity level comparison results from a statistical standpoint; ② S_{sig_cmp} and S_{cst_cmp} derive metric-level outcomes through a voting-like process; ③ S_{non_sig} and $S_{conflict}$ assess whether a specific granularity objectively reflects the overall metric-level voting results, while $S_{non\_obj} = S_{non\_sig} + S_{conflict}$ is further leveraged to *reflect its non-objectivity.* In practice, we calculate the proportion of comparison results that satisfies each of the above dimensions.

Correlation Analysis. We utilize the *Pearson Correlation Coefficient* $r \in [-1, 1]$ [1] for correlation analysis. This coefficient evaluates whether variations in the magnitude of one variable correspond to the variations in the magnitude of another variable, either in the same (positive) or opposite (negative) direction. As we focus on the tool comparison evaluation ability of each metric granularity, our correlation analysis centers on the significance level (p-value) achieved from the significance analysis.

To enhance computational efficiency, we introduce a *significance degree*, denoted as $p' = sign(p)(1-p) \in [-1, 1]$, where p' signifies the strength of significance and sign(p) indicates the comparison result of the means. For instance, in the comparison between Tool A and Tool B, $p' = 0.95$ indicates that Tool A > Tool B is statistically significant with $p = 0.05$, while a p' value of -0.9 denotes that Tool A < Tool B is not statistically significant with $p = 0.1$. Extreme cases arise when $p' = 0$, signifying identical means, and when $p' = 1$ or -1 (corresponding to $p = 0.00$) shows that the variance of both sample groups is 0 but their means can still be differentiated.

Consequently, from 1085 comparisons, we have 1085 samples for each metric granularity. When calculating the correlation coefficient r, since the magnitudes of different metric granularities are exactly the same, r reflects the proportion of changes relatively; specifically, $r = 1$ (-1) indicates that the performance of the two metric granularities is completely consistent (opposite).

Variation Analysis. Examining the variation of the sample data is another way to intuitively quantify randomness. Besides variance, we utilize the *Coefficient of Variation* (CV), which is dimensionless and facilitates comparisons across data sets of different scales [5]. The CV is defined as $CV\% = \frac{STD}{Mean} \times 100\%$, also known as Normalized Standard Deviation. According to [5], a CV exceeding 30 suggests that the experiment is out of control, indicating excessive interference from randomness and resulting in very low acceptability of the experimental outcomes in our context.

2.6 Experimental Protocol

With the purpose of conducting more reliable, comprehensive, and in-depth study, as well as facilitating future evaluation efforts, we propose and release an effective evaluation framework, which has been rigorously tested with the TB-level data of our study.

The workflow of our evaluation framework is illustrated in Figure 1. Initially, we organized the app under test, testing approaches, and various data collectors for GUI testing. The objects for our analysis are derived from these data collectors that monitor tests and extract valuable experimental data. In this study, the collected data includes Activity data for Activity coverage, Jacoco data for fine-grained coverage, Logcat data for faults with exceptions, and ANR data for ANR faults. Given that certain experiments produce TB-level experimental data, a *Data Manager* with a *Database* is required for managing and scheduling. To conduct statistical analysis, we have a *Result Analysis* module consisting of three approaches: *Significance Analysis*, *Correlation Analysis*, *Variation Analysis*, relating to different statistical analysis methods. Tailored for this study, we have analyzers corresponding to the individual RQs, each flexibly employing various underlying analysis methods based on practical demands to achieve effective statistical analysis.

3 STUDY RESULTS AND ANALYSIS

3.1 RQ1: Granularity of Test Metrics

The eight heat maps presented in Figure 2 illustrate the code coverage and fault detection achieved by eight testing approaches across 42 apps, evaluated at all four granularities for each metric. The intensity of the heat in each cell signifies *the net result of significant comparisons against the other seven approaches*, or *net significant score* for short. This is calculated by subtracting the number of approaches for which it was significantly less effective from the number of approaches it outperformed significantly, ranging from -7 to 7. In essence, a cell that is redder indicates that the approach has a higher *net significant score*, i.e., is significantly more effective than more of the other approaches, whereas a bluer cell indicates a lower *net significant score*. Additionally, the number displayed above each cell represents the actual mean code coverage or fault detection result achieved by the approach.

Generally, no approach has achieved significantly high effectiveness across all 42 apps at any of the eight granularities, nor has any approach exhibited consistently low effectiveness across all apps. This observation indicates considerable variability in the performance of different approaches depending on the specific characteristics of each app, underscoring the necessity for a diverse selection of experimental subjects to provide a comprehensive evaluation of the effectiveness of specific Android testing approaches.

In terms of code coverage, as illustrated by the top four heat maps in Figure 2, DQT ranks first undoubtedly, with the highest *net significant score* across all code coverage granularities on 28 apps. This result is anticipated, given that DQT is the latest Android GUI

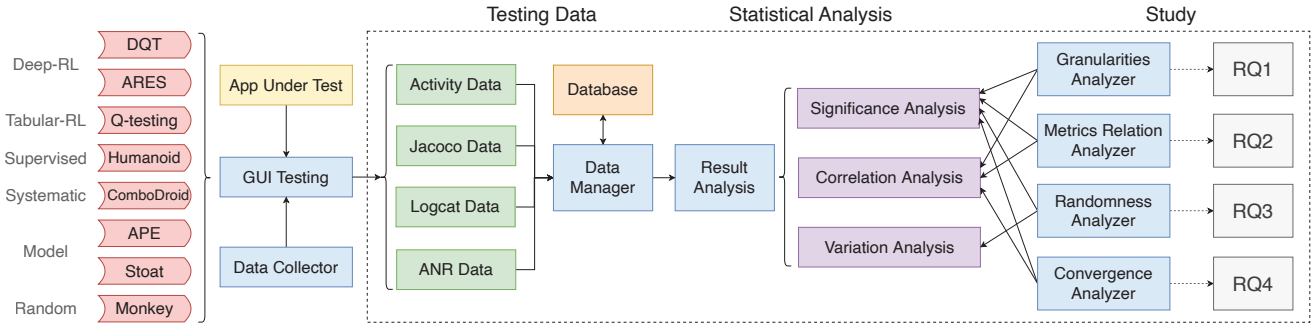

Figure 1: The Workflow of Our Evaluation Framework.

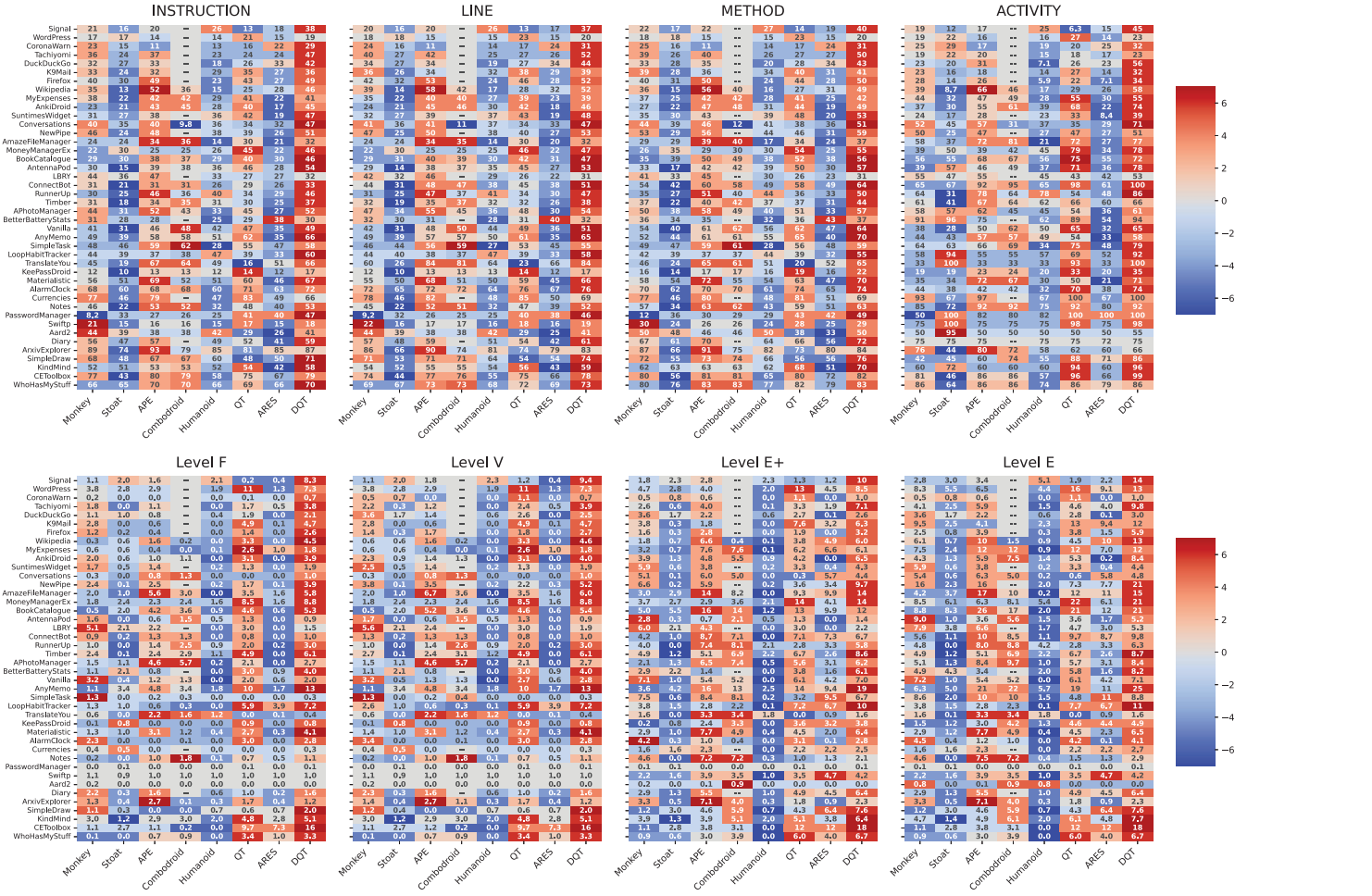

Figure 2: The Average of Code Coverage (Top) and Fault Detection (Bottom) With 8 Testing Approach on AndroTest24.

testing approach leveraging advanced deep reinforcement learning techniques. APE, which achieves the highest on 7 apps, ranks second, and then followed by ComboDroid, QT, Monkey, Humainoid, ARES, and Stoat. In addition, though Stoat fails to achieve the highest *net significant score* at instruction, ELOC, or method levels on any app, it surprisingly achieves the highest at Activity coverage on apps like *LoopHabitTracker* and *Aard2*. This observation prompts a detailed investigation into the correlation of code coverage results across different granularities among various approaches.

These detailed results can be found in Table 4, the left part of which demonstrates the S_{gra_sig}, S_{sig_cmp}, S_{cst_cmp}, S_{non_sig}, $S_{conflict}$ at the four code coverage granularities. The comparison results among the eight approaches at the four coverage granularities have respectively shown significance in 71.3%, 72.1%, 71.8%, and 62.9% of cases. Furthermore, in 47.1% of cases, approximately half of the comparison results, the four granularities exhibit consistent significance. Notably, no conflicting significant results between approaches have been observed at the instruction, ELOC, and method coverage levels whereas the percentage of the non-significant with S_{sig_cmp}, i.e., S_{non_sig}, remains below 2%. Additionally, according to Table 5, our correlation analysis indicates that the significance results across these three granularities exhibit a pairwise coefficient of at least 0.98, which is extremely close to 1. **These observations indicate an extremely strong positive correlation among the three levels of code coverage granularities with respect to the evaluation of coverage significance.** Furthermore, an in-depth analysis focusing on the 10-round *mean value* comparison across these three code coverage granularities, rather than employing the *Independent t-Test*, reveals that among the three groups of 1085 comparison results (see Section 2.5) on instruction, ELOC, and method coverage, 1051, or 96.9%, exhibit consistent outcomes. **This observation suggests that the consistency in significance across instruction, ELOC, and method coverage persists even in mean value comparisons as well.** These findings are noteworthy due to their practical implications, particularly in scenarios where achieving instruction or ELOC coverage is challenging, such as the closed-source commercial apps that are difficult to instrument for fine-grained code coverage statistics. In these cases, utilizing method coverage, which is more readily accessible (e.g., [41]), can serve as the main granularity for code coverage evaluation, which, to some extent, also reflects the effectiveness of testing approaches in terms of both instruction and ELOC coverage.

Table 4: Significance Analysis Results for Metric Granularity

	Code Coverage				Fault Detection			
	INST	ELOC	METH	ACTV	F	V	E+	E
S_{gra_sig}	71.34%	72.07%	71.80%	62.86%	59.72%	61.94%	70.05%	70.51%
S_{sig_cmp}	69.77%				50.51%			
S_{cst_cmp}	47.10%				44.52%			
S_{non_sig}	1.32%	0.13%	0.13%	25.89%	4.56%	0.73%	2.19%	4.20%
$S_{conflict}$	0.00%	0.00%	0.00%	5.02%	0.00%	0.00%	0.00%	0.18%
S_{non_obj}	1.32%	0.13%	0.13%	30.91%	4.56%	0.73%	2.19%	4.38%

Table 5: Correlation Analysis Results for Metrics Granularity

	INST	ELOC	METH	ACTV
INST	1	0.994	0.983	0.678
ELOC	0.994	1	0.989	0.692
METH	0.983	0.989	1	0.709
ACTV	0.678	0.692	0.709	1

(a) Code Coverage

	F	V	E+	E
F	1	0.968	0.656	0.645
V	0.968	1	0.688	0.672
E+	0.656	0.688	1	0.940
E	0.645	0.672	0.940	1

(b) Fault Detection

However, regarding Activity coverage, 30.9% of the experimental results show inconsistent significance performance between Activity coverage and the other three levels of code coverage, out of which around 5.0% even exhibit conflicting significance results. Furthermore, over 70% of these conflicting significance results were found in apps with fewer than 10 Activities. This suggests that the small number of total Activities is likely the primary cause of the bias in Activity coverage statistics. For instance, consider the app *SimpleDraw*, a small-scale Android app containing 8 Activities. It includes an *AboutActivity*, which is controlled by a small and easily overlooked setting option and is the only access to three other activities. Besides, these four Activities are relatively simple activities with few codes. While Monkey and APE effectively explored the remaining four Activities with abundant functions and achieved high instruction, ELOC, and method coverage, they often failed to enter *AboutActivity*. As a result, Monkey and APE exhibit significantly lower Activity coverage while achieving significantly higher instruction, ELOC, and method coverage compared to other approaches. Based on this finding, we suggest that stakeholders in Android testing exercise caution when relying solely on Activity coverage for effectiveness evaluation, especially in scenarios where the experimental subjects contain a limited number of Activities.

The four heat maps at the bottom of Figure 2 present the *net significant score* of fault detection granularities achieved by the eight testing approaches, which can be generally divided into two groups: those on Level F and V faults, and those on Level E+ and E faults. On both Level F and V faults, DQT ranks first, detecting the significantly most faults on 28 apps, followed by QT, APE, Monkey, ComboDroid, Stoat, ARES, and Humanoid. However, on both E+ and E faults, although DQT still ranks first, APE, rather than QT, achieves the second place, followed by ComboDroid, QT and Monkey. This ranking information to some extent implies the gap between the two groups of significance results, which is even clearer in the correlation analysis results presented in the right section of Table 5. In Table 5, a very strong correlation (over 0.9) can be observed between significance results on Level F and V faults, as well as Level E+ and E faults. However, the correlation between the two groups is obviously lower. In detail, significance results on Level F and E+ faults exhibit a correlation of 0.66, markedly lower than the 0.97 between Level F and V faults. A possible explanation for this disparity may be attributed to the nature of the faults themselves. Compared to Level E+ and E faults, which encompass a broader spectrum of fault types, Level F and V faults are generally more severe and challenging to trigger, resulting in more consistent fault-triggering patterns. The eight testing approaches, each characterized by distinct testing strategies, may exhibit varying performance in following these fault-triggering patterns, thereby contributing to the observed variability in the significance results across the four fault levels.

The right part of Table 4 gives an overview of the significance results on fault detection: the comparison results among the eight approaches at the four fault granularities have respectively shown significance in 59.7%, 61.9%, 70.1%, and 70.5% of cases. Besides, consistent significance across the four granularities is observed in 44.5% of cases, which is lower than that observed on code coverage. Even though, almost no conflict significance results were found. Additionally, the S_{non_sig} of each granularity is within 5%, indicating that despite the gap between significance results on Level F, V faults and those on Level E+, E faults, the significance disparities among the four fault granularities are generally not substantial.

Taking the aforementioned findings into comprehensive consideration, we suggest that for stakeholders of Android testing, it would be better to include both Level F (or V) and Level E (or E+) faults for a more comprehensive evaluation of testing approaches in terms of fault detection.

3.2 RQ2: Relations Across Test Metrics

As well-recognized test metrics in Android testing, code coverage, and fault detection are widely adopted for evaluation simultaneously. However, to the best of our knowledge, the correlation between them has never been analyzed. To fill this gap and answer our RQ2, we investigate this correlation.

Table 6: Correlation Analysis Results Across Metrics

		Fault Detection			
		F	V	E+	E
Code Coverage	INST	0.535	0.523	0.486	0.531
	ELOC	0.542	0.527	0.498	0.543
	METH	0.542	0.527	0.497	0.542
	ACTV	0.478	0.471	0.434	0.486

Table 6 presents the correlation between the significance results of the four code coverage granularities (vertical) and those of fault detection granularities (horizontal). The correlation between the results across various code coverage granularities and fault granularities ranges from 0.434 to 0.542, indicating a moderate positive relationship, which is not as strong as that observed between the INST, ELOC, and METH granularities of code coverage.

From a micro perspective analyzing the correlation of individual testing approaches, although DQT demonstrates the highest effectiveness in both code coverage and fault detection, discrepancies between these metrics can still be observed. For instance, on app *WordPress*, DQT detected the most F-level (or V-level) faults with a

net significant score of 5 but achieved the median ELOC coverage among the eight approaches with a *net significant score* of 0. More discrepancies between code coverage and fault detection are evident among the other seven approaches: Monkey, for example, achieves outstanding ELOC coverage on *Swift* (a *net significant score* of 7), its effectiveness in detecting E-level faults was considerably lower (a *net significant score* of -4). Conversely, ARES, despite its poor ELOC coverage on *K9Mail* (a *net significant score* of -4), detected a relatively high number of E-level faults (a *net significant score* of 3).

A possible explanation for these findings is that faults within Android apps tend to interweave with specific functions or modules. For instance, over 40% of the E-level faults found by ARES in *K9Mail* were localized to the module *com.fsck.k9.mail.store.imap*, which implements backend APIs above the IMAP protocol. Despite not achieving extensive app-level coverage like other approaches, ARES conducted a thorough examination of this critical module, uncovering numerous faults. In contrast, APE, despite its superior performance in code coverage at all granularities, detected fewer than half the E-level faults compared to ARES, primarily due to its overlook of this pivotal module.

Anyway, we suggest considering both code coverage and fault detection when evaluating the effectiveness of different testing approaches, especially on large and complex apps.

3.3 RQ3: Test Randomness on Test Metrics

To address RQ3, we investigate the influence of varying test rounds on the evaluation with different test metrics.

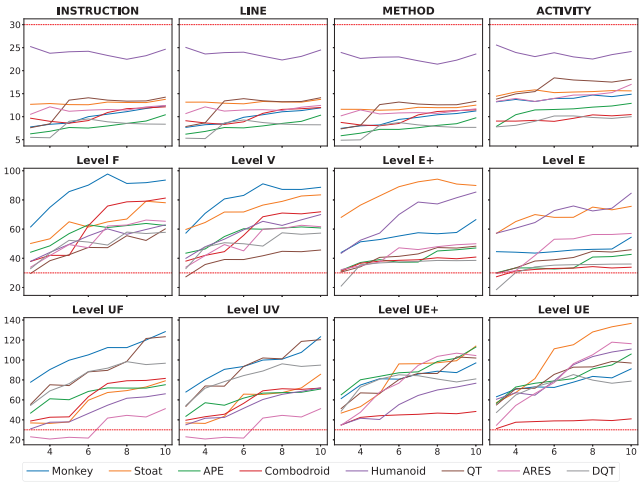

Figure 3: CV of Different Test Rounds.

Figure 3 presents the CV of the eight approaches' experimental results at each metric granularity across different test round configurations. In examining the four line charts at the top of Figure 3, the CV values for the instruction, ELOC, method, and Activity coverage of the seven approaches (excluding Humanoid) range from 4.9% to 18.4%. In detail, from rounds 3 to 6, there is a slight increasing trend in the CV values of these seven approaches, whereas from rounds 6 to 10, the CV values remain relatively stable, albeit with minor fluctuations. Humanoid, however, achieves significantly higher CV values than all these seven approaches. Our investigation reveals that the main reason for the highly variable and unstable performance issues with Humanoid is its reliance on outdated data as a Supervised-Learning-Based testing approach. The data relied on by Humanoid, however, is no longer applicable to modern Android apps, thereby introducing more randomness and uncertainty. Despite this fact, the CV values of Humanoid's code coverage across rounds 3 to 10 are consistently below 30%, a threshold as outlined for CV [5], which is used to estimate whether the experimental results are out of control. Notably, all CV values across various code coverage granularities remain markedly lower than this threshold, thereby implying a state of relative stability in the utilization of code coverage for effectiveness evaluative purposes.

However, in regard to the four line charts in the middle of Figure 3, almost all eight approaches manifest a gradual increase in CV values of their experimental results across all fault detection granularities. Moreover, with test rounds ranging from 4 to 10, all eight approaches manifest CV values for their fault detection results surpassing the 30% threshold. Noteworthy is the observation that the highest CV value in fault detection, reaching 97.8%, is recorded with Monkey on Level F fault, which is mainly due to the Random-Based nature of Monkey.

In addition, we conduct a more in-depth analysis of another frequently-used adjusted granularity for fault detection: Unique Faults [25, 33, 40], which denote the number of faults that can *only be detected by a specific approach among all compared approaches*. The CV values of the unique faults detected at Levels F (UF), V (UV), E+ (UE+), and E (UE) by eight approaches are shown as the four line charts at the bottom of Figure 3. Unfortunately, no signs of convergence but an increasing trend has been observed on the CV values of the unique faults' results. In addition, these CV values observed are generally higher than those recorded for the original fault results with the highest CV value of unique faults, reaching 136.6%, recorded with Stoat on Unique Level E (UE) faults.

For the extremely high and progressively increasing CV values observed in the fault detection and unique fault results, a possible explanation may be that, despite the configuration of 10 test rounds—the maximum test rounds that have ever been taken in existing Android testing studies—the convergence of variability in fault detection results has yet to commence. The minimal test rounds required to attain stable fault detection results, however, remain unexplored and we left it as our future work, considering that existing experiments with 10 test rounds have already consumed us 727 CPU days and a time duration over six months.

These findings align with our conjecture in addressing RQ2, that the faults tend to be interwoven within specific functions or modules, potentially leading to a heightened occurrence of biased results compared to evaluation on code coverage.

Evaluating testing approaches on code coverage is significantly more stable compared to that on fault detection. In addition, despite the configuration of maximum test rounds presented in previous research, a significant increment of variability can still be observed in the fault detection results. Based on these findings, we notice Android testing stakeholders to be careful in drawing conclusions from comparative analyses of mean fault numbers, especially mean numbers of unique faults, given the inherent instability associated with this metric.

3.4 RQ4: Test Convergence on Test Metrics

With the convergence phenomenon observed in prior studies [7, 25, 33, 40, 41], RQ4 investigates the convergence of each test metric over time, including both code coverage and fault detection.

Figure 4 depicts the average time required by the eight approaches to reach n% of the final 3-hour results across the four granularities of code coverage (the top four charts) and fault detection (the bottom four charts).

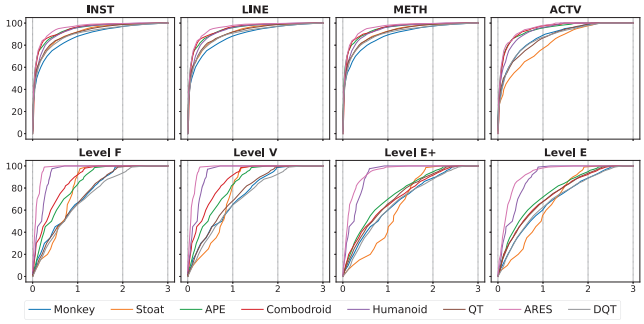

Figure 4: Time Required to Reach N% of the Final 3.0h Result

In terms of instruction, ELOC, and method coverage, a rapid initial increase can be observed for all approaches, followed by a gradual plateauing. The curves of all eight approaches are closely clustered together, indicating consistent growth trends observed on all eight approaches across these three coverage granularities. Nevertheless, a discernible delay in the convergence of Activity coverage is evident compared to the prior three. Despite this, it is noteworthy that all eight approaches reach 80% of the final coverage within 1.1 hours on average across all coverage granularities, with the achievement of 90% coverage within 1.5 hours. In addition, a rapid coverage convergence can be found on approaches such as ARES, APE, and ComboDroid, which reach 90% code coverage at all coverage granularities within 0.58 hours.

Fault detection results, as demonstrated in the four line charts at the bottom of Figure 4, present significant more variability compared to the code coverage results. All eight methodologies demonstrate noticeably faster growth in detecting faults at Levels F and V compared to those at Levels E+ and E. This observation is logical given that Fatal Errors and Core Vitals are critical errors with typically smaller bases, thus facilitating convergence. Nevertheless, these approaches still reveal over 80% of Levels E+ and E faults within 1.6 hours and reveal 90% within 2.1 hours. Surprisingly, ARES and Humanoid reveal 100% faults across four Levels faults within 1.7 hours, i.e., merely 57% of the total test time. This interesting observation of the two approaches' fault detection results may explain this phenomenon: the two approaches' exploration depth across the 42 apps appears relatively shallow, as evidenced by their coverage performance. Hence, although they do reveal quite a number of faults in these shallow app modules or functions, their limited depth in the app-level exploration hinder their fault detection capability in the later stages.

Accordingly, we conducted a general correlation analysis between the results of approach comparison using the data at each time point and the results obtained with the final 3-hour data, as shown in Table 7. The 2-hour results exhibit a very high positive correlation with the final results (+0.969 ∼ +0.979), which is nearly perfect (+1), for both code coverage and fault detection. **These observations indicate that for most testing approaches on most Android apps, a moderate test duration is sufficient for drawing preliminary conclusions on their overall effectiveness in both terms of code coverage and fault detection.** In light of this discovery, we suggest that stakeholders engaged in Android testing, particularly approach developers, consider configuring their effectiveness experiments with a moderate test duration, such as two hours, when adjusting their approaches under developing on a number of Android apps. This suggestion, we believe, holds promise for enhancing the efficiency of testing approach development.

Table 7: Correlation Analysis Between the Results at Different Time Points and the Final 3.0h Results

	Code Coverage				Fault Detection			
	INST	ELOC	METH	ACTV	F	V	E+	E
0.5h_3.0h	0.817	0.820	0.819	0.717	0.798	0.797	0.729	0.727
1.0h_3.0h	0.912	0.914	0.915	0.823	0.903	0.897	0.849	0.861
1.5h_3.0h	0.953	0.955	0.958	0.879	0.947	0.943	0.908	0.923
2.0h_3.0h	**0.977**	**0.979**	**0.979**	**0.969**	**0.975**	**0.973**	**0.969**	**0.971**
2.5h_3.0h	0.994	0.995	0.995	0.992	0.993	0.993	0.991	0.991

4 THREATS TO VALIDITY

Internal Threats. The primary internal threats stem from the screening of metric granularities and the processing of raw Logcat data. To address these issues, we analyzed the metric granularities adopted by previous studies, and referred to Google's official definitions, dividing code coverage and fault detection into four granularities each. For Logcat processing, three authors independently applied filtering, deduplication, and categorization, with mutual proofreading to ensure more reliable outcomes.

External Threats. External threats mainly arise from the selection of experimental apps and approaches. To mitigate these threats, we combine more than ten open-source benchmarks proposed by previous works, filtering out outdated apps to construct our app benchmark, AndroTest24. For comparative analysis, we systematically selected 8 well-known SOTA Android GUI testing approaches representing various categories and methodologies.

Another notable external threat comes from the data loss of ComboDroid [40] on 13 apps due to widely recognized scalability issues [25, 33] of static analysis. Given that the 42 apps were systematically selected from previous research studies, excluding these 13 apps would not only introduce substantial app selection bias but also exacerbate the issue of data loss. For example, there would be a loss of $13 \times C_7^2 = 273$ (23.2% of the total) pairs in significance analysis. To alleviate such issues, we opted to preserve the original results to the greatest extent possible, thereby reducing the data loss to a more manageable level–for example, a loss of $13 \times 7 = 91$ (only 7.7% of the total) pairs in significance analysis. Moreover, the remaining 29 apps still span 17 of the 19 categories represented by the complete app set, maintaining a broad app diversity.

5 FRAMEWORK

Our framework has demonstrated high practical effectiveness when applied to TB-level data, as evident in our study. It is unique in the

out-of-the-box functionality with rigorous statistical analysis for test metrics and efficient data management integration for large-scale, multi-type testing data. Both the study data and the framework code are publicly available at https://github.com/Yuanhong-Lan/AndroTest24 and ready for immediate use.

We believe this framework holds substantial value for future mobile testing evaluations, including not only tool comparisons but also tool development and enhancement. For example, testing tool developers can perform more statistically rigorous comparisons and advance new tools, while testing tool users can easily select applicable testing tools with the framework. Moreover, the applicability of our framework extends beyond mobile testing to other domains such as web testing and API testing, which face similar challenges in metric evaluation. By adopting and expanding, our framework could enlighten and support additional areas as well.

6 RELATED WORKS

In this section, we will delineate both the empirical studies and automated testing approaches in Android GUI testing.

6.1 Empirical Studies

In 2015, Choudhary et al. [7] conducted a seminal study that systematically compared Android GUI testing approaches and evaluated their strengths and weaknesses. To their surprise, they discovered that rudimentary random strategies such as Monkey [16] and Dynodroid [28] exhibited significantly superior performance compared to more sophisticated approaches at the time. Additionally, they introduced an open-source app benchmark, AndroTest, which has since achieved widespread recognitions [10, 24, 25, 30, 33, 35, 36].

Subsequently, Zeng et al. [42] and Zheng et al. [43] embarked on an industrial case study on the application of Monkey [16] on *Wechat* [38]. Their study revealed various limitations and proposed several improvements. On the other hand, considering the simpler nature of open-source apps, Wang et al. [41] presented an empirical investigation under a set of industrial apps and suggested combining different approaches for better performance.

Later, Behrang et al. [4] discovered that, apart from exploration strategies, other factors such as input files and system events contribute a lot to the testing performance. Through an in-depth study, they pinpointed seven general categories of limitations and endeavored to address them for coverage improvements. In addition to code coverage, fault detection is also a key focus. Su et al. [37] introduced the THEMIS benchmark consisting of reproducible crash bugs where a considerable gap was discovered between state-of-the-art testing approaches and real-world bugs.

In contrast to prior studies, our study reveals a remarkable number of unreliable and conflicting evaluation results attributable to an imperfect evaluation system. To mitigate such issues, we conducted the first comprehensive statistical analysis of evaluation metrics in Android GUI testing. Our study yielded several notable findings, practical recommendations, and an effective evaluation framework.

6.2 Android GUI Testing Approaches

Following previous studies [4, 25, 33, 35], existing Android GUI testing approaches are generally classified into four main categories according to the major testing strategy they rely upon.

Random-Based Testing. These approaches [16, 28] generate pseudo-random test inputs rapidly. Monkey [16], developed by Google and integrated into the Android platform, is acknowledged as one of the most popular testing approaches in the industry.

Model-Based Testing. The fundamental idea underlying model-based approaches [2, 3, 6, 20, 24, 27, 32, 36] is the creation of a model representing the application under test, which is then employed to generate test inputs. These models are typically constructed using either static analysis or dynamic analysis techniques, such as finite-state machines. Stoat [36], for instance, combines static and dynamic analysis to reverse engineer stochastic models of target applications for test guidance. Conversely, APE [20] dynamically refines the model of the target application using runtime information instead of relying solely on a static model.

Systematic Testing. With more sophisticated techniques such as symbolic execution and evolutionary algorithms, systematic approaches [10, 12, 29–31, 40] generate tailored test inputs for specific objectives. TimeMachine [10] proposes time-travel testing and evolves a population of states that could be saved and resumed when needed via virtualization. By contrast, ComboDroid [40] systematically combines short use cases based on the data dependencies to achieve longer, meaningful, and effective test paths.

Machine-Learning-Based Testing. The rapidly advancing machine learning technologies have been widely applied in Android GUI testing. A significant number of approaches [8, 21–23, 26] capitalize on supervised learning, leveraging historical data to inform their testing processes. For instance, Humanoid [26] trains a deep neural network model on extensive human interaction traces, enabling the generation of effective human-like test inputs.

In contrast, without requiring pre-training a model in advance, recent approaches [9, 25, 33–35] benefit from reinforcement learning, learn from runtime testing interactions and dynamically optimize their testing policies. Q-testing [33], based on Tabular RL, utilizes a Q-table guided by a curiosity strategy bolstered by a scenario division model to conduct flexible and adaptive tests. ARES [35] and DQT [25] further advance this frontier by introducing Deep RL to enhance the learning capabilities of the guidance model. While ARES showcases the potential of Deep RL, DQT further harnesses the profound ability of the testing knowledge-sharing mechanism.

7 CONCLUSION

Through an extensive survey, we identified notable evaluation challenges in mobile testing and conducted the first comprehensive statistical analysis of the main existing Android GUI testing metrics as well as their configurations. With extensive experiments across 8 SOTA testing approaches on 42 diverse well-recognized open-source apps, advanced statistical methods were novelly applied for in-depth significance, correlation, and variation analysis. Our study reached notable findings and practical suggestions, as well as an effective evaluation framework that is publicly available, to empower and enhance future mobile testing evaluations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This research was supported by the National Natural Science Foundation of China under Grant Nos. 62372227 and 62032010.

REFERENCES

[1] 2018. Correlation coefficients: Appropriate use and interpretation. *Anesthesia and analgesia* 126, 5 (1 May 2018), 1763–1768. https://doi.org/10.1213/ANE.0000000000002864

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, Michael Goedicke, Tim Menzies, and Motoshi Saeki (Eds.). ACM, 258–261. https://doi.org/10.1145/2351676.2351717

[3] Young Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 238–249. https://doi.org/10.1145/2970276.2970313

[4] Farnaz Behrang and Alessandro Orso. 2020. Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for Android. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 1066–1077. https://doi.org/10.1145/3324884.3416567

[5] Charles E. Brown. 1998. *Coefficient of Variation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 155–157. https://doi.org/10.1007/978-3-642-80328-4_13

[6] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 623–640. https://doi.org/10.1145/2509136.2509552

[7] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 429–440. https://doi.org/10.1109/ASE.2015.89

[8] Faraz Yazdani Banafshe Daragh and Sam Malek. 2021. Deep GUI: Black-box GUI Input Generation with Deep Learning. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 905–916. https://doi.org/10.1109/ASE51524.2021.9678778

[9] Christian Degott, Nataniel P. Borges Jr., and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 296–306. https://doi.org/10.1145/3293882.3330569

[10] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 481–492. https://doi.org/10.1145/3377811.3380402

[11] EclEmma. 2024. *JaCoCo Java Code Coverage Library*. Retrieved April 6, 2024 from https://www.eclemma.org/jacoco/

[12] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 419–429. https://doi.org/10.1145/3238147.3238225

[13] Google. 2023. *Android vitals*. Retrieved October 11, 2023 from https://developer.android.com/topic/performance/vitals

[14] Google. 2023. *Crashes*. Retrieved September 28, 2023 from https://developer.android.com/topic/performance/vitals/crash

[15] Google. 2023. *Run apps on the Android Emulator*. Retrieved May 1, 2023 from https://developer.android.com/studio/run/emulator

[16] Google. 2023. *UI/Application Exerciser Monkey*. Retrieved April 12, 2023 from https://developer.android.com/studio/test/other-testing-tools/monkey

[17] Google. 2024. *ANRs*. Retrieved January 3, 2024 from https://developer.android.com/topic/performance/vitals/anr

[18] Google. 2024. *Logcat command-line tool*. Retrieved January 3, 2024 from https://developer.android.com/studio/command-line/logcat

[19] Google. 2024. *Support for ARM binaries on Android 9 and 11 system images*. Retrieved April 30, 2024 from https://developer.android.com/studio/releases/emulator#support_for_arm_binaries_on_android_9_and_11_system_images

[20] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 269–280. https://doi.org/10.1109/ICSE.2019.00042

[21] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 269–282. https://doi.org/10.1145/3236024.3236055

[22] Nataniel P. Borges Jr., Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018*, Christine Julien, Grace A. Lewis, and Itai Segall (Eds.). ACM, 133–143. https://doi.org/10.1145/3197231.3197243

[23] Yavuz Köroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-Based Exploration of Android Applications. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 105–115. https://doi.org/10.1109/ICST.2018.00020

[24] Duling Lai and Julia Rubin. 2019. Goal-Driven Exploration for Android Applications. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 115–127. https://doi.org/10.1109/ASE.2019.00021

[25] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 71:1–71:13. https://doi.org/10.1145/3597503.3623344

[26] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1070–1073. https://doi.org/10.1109/ASE.2019.00104

[27] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 135:1–135:5. https://doi.org/10.1145/3551349.3559505

[28] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 224–234. https://doi.org/10.1145/2491411.2491450

[29] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 599–609. https://doi.org/10.1145/2635868.2635896

[30] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 94–105. https://doi.org/10.1145/2931037.2931054

[31] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. SIG-Droid: Automated system input generation for Android applications. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 461–471. https://doi.org/10.1109/ISSRE.2015.7381839

[32] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 559–570. https://doi.org/10.1145/2884781.2884853

[33] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 153–164. https://doi.org/10.1145/3395363.3397354

[34] Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. 2023. Badge: Prioritizing UI Events with Hierarchical Multi-Armed Bandits for Automated UI Testing. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 894–905. https://doi.org/10.1109/ICSE48619.2023.00083

[35] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep Reinforcement Learning for Black-box Testing of Android Apps. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 65:1–65:29. https://doi.org/10.1145/3502868

[36] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of*

Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 245–256. https://doi.org/10.1145/3106237.3106298

[37] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 119–130. https://doi.org/10.1145/3468264.3468620

[38] Tencent. 2024. *Connecting a billion people with chat, calls, and more.* Retrieved March 6, 2024 from https://www.wechat.com/en/

[39] Raoul R. Wadhwa and Raghavendra Marappa-Ganeshan. 2023. *T Test.* StatPearls Publishing, Treasure Island (FL).

[40] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: generating high-quality test inputs for Android apps via use case combinations. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 469–480. https://doi.org/10.1145/3377811.3380382

[41] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018,* Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 738–748. https://doi.org/10.1145/3238147.3240465

[42] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: are we really there yet in an industrial case?. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 987–992. https://doi.org/10.1145/2950290.2983958

[43] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017.* IEEE Computer Society, 253–262. https://doi.org/10.1109/ICSE-SEIP.2017.32