Received: 1 November 2022 | Revised: 27 September 2023 | Accepted: 4 November 2023

DOI: 10.1002/smr.2640

SPECIAL ISSUE - EMPIRICAL PAPER

Journal of Software: Evolution and Process WILEY

GUI testing of Android applications: Investigating the impact of the number of testers on different exploratory testing strategies

Sergio Di Martino | Anna Rita Fasolino | Luigi Libero Lucio Starace | Porfirio Tramontana

Department of Electrical Engineering and Information Technology (DIETI), Università degli Studi di Napoli Federico II, Naples, Italy

Correspondence
Luigi Libero Lucio Starace, Università degli Studi di Napoli Federico II, Naples, Italy.
Email: luigiliberolucio.starace@unina.it

Funding information
ENACTEST Project (European iNnovation AllianCe for TESting educaTion)- ERASMUS-EDU-2021-PI-ALL-INNO, Grant/Award Number: 101055874; PNRR MUR Project FAIR (Future Artificial Intelligence Research), Grant/Award Number: PE0000013

Abstract

Graphical user interface (GUI) testing plays a pivotal role in ensuring the quality and functionality of mobile apps. In this context, exploratory testing (ET), a distinctive methodology in which individual testers pursue a creative, and experience-based approach to test design, is often used as an alternative or in addition to traditional scripted testing. Managing the exploratory testing process is a challenging task that can easily result either in wasteful spending or in inadequate software quality, due to the relative unpredictability of exploratory testing activities, which depend on the skills and abilities of individual testers. A number of works have investigated the diversity of testers' performance when using ET strategies, often in a crowdtesting setting. These works, however, investigated ET effectiveness in detecting bugs, and not in scenarios in which the goal is to generate a re-executable test suite, as well. Moreover, less work has been conducted on evaluating the impact of adopting different exploratory testing strategies. As a first step toward filling this gap in the literature, in this work, we conduct an empirical evaluation involving four open-source Android apps and 20 masters students that we believe can be representative of practitioners partaking in exploratory testing activities. The students were asked to generate test suites for the apps using a capture and replay tool and different exploratory testing strategies. We then compare the effectiveness, in terms of aggregate code coverage that different-sized groups of students using different exploratory testing strategies may achieve. Results provide deeper insights into code coverage dynamics to project managers interested in using exploratory approaches to test simple Android apps, on which they can make more informed decisions.

KEYWORDS
Android applications, capture and replay, exploratory testing, GUI testing

1 | INTRODUCTION

Mobile apps have become pervasive and are increasingly involved in many economic, social, and recreational aspects of our daily lives. According to a recent report,[1] 4 billion smartphone users have downloaded 230 billion apps in 2021, with an app store spend of $170 billion. In such a competitive market, it is fundamental to adequately test mobile apps, as high-quality apps have a much higher chance of being well-received by users and thus of being profitable.[2]

As most of the interaction with a mobile app is performed through its graphical user interface (GUI), a great deal of testing efforts are focused on GUI-level interactions, and aim at verifying that the application under test (AUT) behaves as expected in response to sequences of input GUI events.

As for the generation of these sequences of GUI events, Exploratory Testing (ET)[3] is a widely-used strategy, focusing on a creative, experience-based approach in which test design, execution, and learning are carried out iteratively by testers, and the designing of new tests is mostly based on the knowledge resulting from previously-executed tests.[4] Itkonen et al., on the basis of the results of an industrial field study, concluded that the ET approach could be effective even when performed by testers with limited experience.[5]

ET can be supported by capture and replay (C&R) tools (e.g., Robotium Recorder[6] and Espresso Test Recorder[7]) that automatically generate repeatable tests from sequences of user interactions with the AUT, thus '*capturing*' (*recording*) real usage scenarios, without requiring advanced testing/programming skills. Subsequently, these records/scripts can be replayed multiple times on different devices to reduce the overall testing efforts.

The simplest approach to exploratory GUI testing of Android apps with C&R techniques is the one in which testers have no previous knowledge neither on the AUT nor on its functional requirements but are rather free to define tests, that is, sequences of user interactions, based on their sensibility. Such an approach is referred to in the literature as *uninformed* ET (UET).[8] Alternatively, testers can be driven not only by their own sensibility but also by additional information about the source code of the AUT and by the code coverage achieved by previous tests. This approach is referred to in the literature as *informed* ET (IET).[8] UET and IET approaches are also often used in combination in the test design process. Indeed, it is common for practitioners to begin the testing activities using an uninformed approach, and subsequently refine the test suite by switching to an Informed strategy.[9] We refer to such a combination featuring a preliminary UET phase followed by an IET phase as U+IET.

New perspectives for ET raised in the last years with the diffusion of crowdsourced testing[10–13] (or crowdtesting). The idea behind crowdtesting is to use a "*crowd*" of several people, typically recruited using dedicated online services,[14–19] to carry out testing tasks on-demand and relatively inexpensively.[20] The results of these testing tasks can then be aggregated in some meaningful way to increase the overall confidence that the system under test behaves as expected, possibly in a more cost-effective and timely way than traditional testing processes.

Crowdsourced testing has proven to be particularly effective, especially in the mobile applications domain,[10,11,21] and has been adopted by an increasing number of software organizations including Microsoft, Google, PayPal, and Uber,[22] to test their mobile apps. In this field, crowdtesting activities combine well with ET strategies. Indeed, exploratory approaches can greatly benefit from the diverse backgrounds and skill sets of the different crowdworkers, resulting in overall more reliable testing results.[23] Moreover, by requiring crowdtesters to record their interactions with the AUT by means of capture and replay tools, it is also possible to obtain a re-executable test suite to be used in future regression testing activities.

When applying exploratory testing, well-known software project management challenges, such as determining the best trade-off between testing costs and software quality, are particularly critical, due to the complexity of mobile apps and the relative unpredictability of exploratory testing processes.[23] Indeed, allocating too few testers might result in inadequate software quality levels, while allocating too many of them can lead to overspending and hinder the cost-effectiveness of the testing process. As an example of this, a recent empirical evaluation by Wang et al[23] in a crowdtesting context found that, in a popular Chinese crowdsourcing platform, more than 30% of the spending to recruit additional testers was wasted, that is, it did not lead to discovering more defects. The decision on how much resources should be allocated is further complicated by the possibility of employing different exploratory testing strategies, that is, UET, IET, or combinations thereof such as U+IET. Indeed, a project manager might be puzzled in deciding which strategy to adopt in an exploratory testing task, under fixed budget constraints: *Is it better to allocate a greater number of testers without prior knowledge of the AUT, or is it better to allocate a smaller number of testers to carry out also more time-consuming Informed exploration activities?*

A number of works in the literature have proposed decision support systems aimed at assisting practitioners in effectively managing the exploratory testing process, mostly in crowdtesting scenarios, by predicting when to stop[23,24] or by recommending how many and which testers should be recruited.[25] These works, however, consider a scenario in which the goal is to detect faults, and not to also produce a re-executable test suite adequately stressing the AUT, which could be used in subsequent regression testing activities. Moreover, they do not take into account different exploratory strategies. To the best of our knowledge, there exists no work in the literature aimed at supporting the managerial choice of deciding how many testers should be allocated, and which exploratory strategy should be used, in a scenario in which the goal is to obtain re-executable test suites.

In this paper, we empirically investigate the influence of the number of allocated testers on the effectiveness of GUI test suites for Android applications obtained using different exploratory strategies. To this end, we expand on the results of an empirical study that we recently published,[8] involving 20 computer engineering master students of an Italian University, enrolled in a software engineering course. The students were asked to produce a re-executable test suite for each of four open-source Android applications using a C&R tool, firstly with a UET strategy, and subsequently with a U+IET approach. In that study, results highlighted that the entire group of 20 testers could easily achieve the maximum possible code coverage, but significant differences existed in the effectiveness of the test suites produced by single students.

In this work, under the assumption that students can be considered as representatives of practitioners partaking in exploratory testing activities,[26,27] we compare the effectiveness, in terms of aggregate LOC and branch coverage, that different-sized groups of students, using different exploratory testing strategies, would have achieved on the four considered applications.

The results of our study provide an initial exploration of the relationship between different exploratory testing strategies and GUI testing. The findings suggest that exploratory strategies and C&R tools can be a valuable approach for generating re-executable test suites for mobile applications. These results motivate further investigations, which could involve larger and more complex applications, as well as a wider range of participants, also in a crowdtesting context. Additionally, the insights gained from our study may be useful for project managers who are interested in utilizing these approaches for simple Android applications. Lastly, we make all the experimental data (i.e., the test suites generated by each of the students and the source code of the Android applications), as well as all the scripts that we implemented to carry out the analyses, freely available to the public[28] for replication purposes.

The paper is structured as follows. In Section 2, we give some background definitions of exploratory testing, and we present an overview of the state of the art with particular focus on empirical studies investigating the application of exploratory testing strategies. In Section 3, we describe the empirical study that we conducted in terms of research questions, objects, subjects, procedures and metrics. In Section 4, we present and discuss the results of our evaluation, while in Section 5, we discuss possible threats to validity. Finally, in Section 6, we give some final remarks and discuss some possible future research directions.

2 | BACKGROUND AND RELATED WORKS

In this section, we give some background notions on exploratory testing, and we present an overview on the state of the art with a particular focus on empirical studies investigating the effectiveness of exploratory testing and the difference.

2.1 | Exploratory testing

Exploratory testing (ET)[3,29] is a test design strategy that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project.[30]

As witnessed by a number of empirical studies,[5,30–32] exploratory testing is largely used in practice and recognized as fundamental, especially in manual testing activities.[30]

Nonetheless, thanks to *capture and replay* tools that are capable of recording manual interactions to generate re-executable test scripts, exploratory testing can be applied not only to search for bugs manually but also to generate a re-executable test suite for GUI-based applications.

A similar approach has been recently investigated in the context of mobile application testing by Di Martino et al..[8] In that work, the authors formalize two different approaches to exploratory testing, namely, *uninformed* ET (UET) and *Informed* ET (IET).

UET involves conducting testing activities without any specific prior knowledge or guidance about the application being tested. Testers engage in the testing process with little to no preconceived notions, relying on their intuition, creativity, and investigative skills to explore the system. They explore various functionalities, user interfaces, and data inputs to discover potential defects, vulnerabilities, or unexpected behaviors. The UET approach can be used even by testers with no programming or Software Engineering experience and can be effective in uncovering issues that might not be apparent or anticipated based on prior knowledge or documentation.

IET, on the other hand, involves conducting testing activities with a certain level of knowledge, information, or guidance about the application being tested. This approach typically incorporates requirements documentation, design specifications, access to the source code of the AUT and/or to code coverage reports. Testers engage in exploratory testing while leveraging this information to guide their exploration, prioritize areas of interest, and identify potential risks or critical functionalities to focus on. IET allows testers to approach the system with a targeted mindset, enabling them to focus their efforts more effectively. However, using an IET approach requires testers with some programming and Software Engineering experience, and is generally more time-consuming.

It is also possible for developers to leverage both UET and IET approaches to design effective test suites. Indeed, as witnessed also in a recent survey by Aniche et al.,[9] it is common for testers to begin with an initial UET phase. This approach allows testers to gain a comprehensive understanding of the AUT, identify potential areas of risk, and explore the software from a fresh and unbiased perspective. By starting with an uninformed approach, testers can uncover unexpected issues, discover undocumented functionalities, and explore the AUT without the influence of preconceived notions or prior knowledge. The uninformed phase allows testers to explore the AUT in a more holistic fashion before transitioning into the Informed phase, where testers can apply explicit guidance to drive the testing activities. This two-step approach, which we refer to as **U+IET**, can combine the benefits of both UET and IET, enhancing the effectiveness of the testing process.[9]

2.2 | Empirical studies on tester performance in exploratory testing

The diversity of testers' performance has been the subject of numerous empirical studies in the past,[33–36] with the aim of comparing the capability of students and professionals in fault-finding tasks with those of testing techniques or automatic tools.

In a more recent study by Mao et al,[37] an experiment involving 434 testers in the context of a crowdtesting experiment carried out on the Amazon Mechanical Turk platform showed that crowdtesters were able to achieve greatly varying results, complementary to the ones of automatic tools; thus, the use of tests generated by crowdtesters may represent a valid starting point for search-based approaches.

Due to this variability, the selection of more than one tester in exploratory testing contexts generally provides better effectiveness but with an increase in costs. The trade-off between the number of testers (and thus costs) and achieved testing effectiveness has been studied in several works, in particular in the context of crowdtesting. The problem of the optimal selection of a set of crowd workers in order to maximize the coverage of test requirements, and to minimize the number of testers (and the consequent cost) is addressed in many papers[20,38–43] by proposing different search-based approaches. These approaches were tested on a series of historical data on the effectiveness of crowd workers who worked on the Baidu platform. These data show how there is a significant difference in terms of effectiveness between crowd workers and how it can be related to their experience measured from their performance on past tasks solved in the same platform. They also show how the growth trend of test effectiveness is slower and slower as the number of testers independently dealing with the same test task increases. Kamangar et al,[44] instead, try to correlate the test effectiveness of crowd workers to their personality type classified by a psychological approach.

Several other papers in the literature directly studied the relationship between the number of testers and the overall effectiveness of the produced tests. In the context of usability testing, Nielsen et al[45] evaluated the influence of the number of testers and their experience (distinguishing by students or professionals) on the overall capability of finding usability issues. They found that sets of five students were able, on average, to find 50% of the issues, while sets of 14 students were able to find 75% of issues. On the other hand, sets of five professionals were able, on average, to find 90% of issues, while the overall set of 15 professionals were able to find all the usability issues. Also in the context of usability testing, Sears et al[46] found that the percentage of found issues increased from 41% to 61% when the number of testers was increased from 2 to 5.

In the mobile apps domain, a pilot study presented by Wang et al[23] on the basis of data from the Baidu crowdtesting platform observed three distinct phenomena: large variation in bug detection rapidity and cost among different applications, decreasing bug detection rates over time and plateau effect of bug finding curve. They proposed iSense,[24] an estimation technique to evaluate the optimal trade-off between testing effectiveness (in terms of the number of different bugs detected) and cost. In addition, an analysis of the same dataset reported in[47] found that there is a large amount of similar bug reports finding the same bugs, confirming the previously observed phenomena.

All the works on mobile apps crowdtesting in the literature, however, have investigated scenarios in which the goal is to detect faults, and not to also produce a re-executable test suite using capture and replay tools. Moreover, to the best of our knowledge, no other work in the literature takes into account the possible impact of using different exploratory strategies (i.e., *uninformed* and *Informed* approaches, or combinations thereof such as U+IET).

3 | THE EMPIRICAL STUDY

The aim of this paper is to investigate the effectiveness of test suites generated by different-sized non-communicating groups of testers with capture and replay (C&R) tools and with different exploratory testing strategies. In this section, we start by presenting the research questions that we investigate in this study. Then we detail the evaluation we conducted, by describing the involved subjects (i.e., the testers), the tools, the AUTs, the metrics that we considered when comparing the effectiveness of test suites, and the experimental procedure.

3.1 | Research questions

To investigate the impact of the number of involved testers on the effectiveness of the GUI test suites generated with C&R techniques and different exploratory testing strategies, we consider the following research questions.RQ1 What is the effect on code coverage of adding additional workers to non-communicating groups of testers using a UET (resp., U+IET) strategy?

RQ2 How do test suites generated by different-sized groups of testers with a UET strategy compare, in terms of code coverage, to those generated by different-sized groups of testers with a U+IET strategy?

RQ3 Under a fixed budget, how do test suites generated by testers using a UET, U+IET, and mixed approaches compare, in terms of code coverage?

3.2 | Subjects

Our study involved a set of students attending the Advanced Software Engineering course held by one of the authors for the Computer Engineering Laurea degree course of the University of Naples Federico II. Twenty students were enrolled in the experiments. They all had a Computer Engineering Bachelor Degree and a common background in Java programming and software engineering. As the advanced software engineering course is mainly focused on testing, they received several lectures on testing techniques (including functional and coverage-based testing techniques), testing automation, JUnit, GUI testing, and C&R techniques. In addition, they attended several lectures on Android programming basics (the final assignment for the course required them to develop an Android app) and were all Android phone users. Finally, a specific lecture on C&R tools for Android testing was presented.

3.3 | Objects

The selected C&R tool for this experiment was Robotium Recorder.[48] The main feature of this tool is the possibility to record (capture) the interactions of a user with an Android device in the context of an Android app and to generate a corresponding re-executable JUnit test case, exploiting JUnit and the Robotium library.[6] The tool can be straightforwardly integrated with IDEs such as Eclipse and Android Studio.

We used the well-known Emma[49] tool to measure the LOC coverage achieved by the test cases at source code level (with respect to the Executable Lines of Code metric). According to the operative definition provided by the Emma reference manual,[50] an Executable LOC is a line of source code having at least a corresponding statement in the bytecode (e.g., comments are lines of code that do not yield executable LOCs). To measure branch coverage, we instrumented the AUTs, inserting probes in each branch of their Java source code. Such probes produce messages on the standard log output of Android apps, which we captured using the Logcat tool. The instrumented versions of the applications are available in the replication package.

3.3.1 | The applications under test

Well-known limitations of C&R tools are related to their difficulty in registering and reproducing pre-conditions to the test case execution, which can be related to third-party remote resources (e.g., remote databases or web services, including authentication services). In addition, C&R tools may generate unstable test cases in presence of multi-threaded applications and dependencies on race conditions.

We have selected a set of four small-sized Android applications that are single-threaded and that have no dependencies on third-party services. They are all open-source apps (their source codes are also available on the F-Droid[51] repository or on GitHub) and were also used in other empirical studies on Android GUI testing.[52–54]

Table 1 reports, for each of the selected apps, its name and version, as well as the total number of classes, activities, methods, LOCs, Executable LOCs, and branches. We also point out that the number of classes reported in Table 1 includes both statically defined and anonymous classes, including listener classes related to GUI widgets.

More in detail, MunchLife (A1) is a counter of the score achieved in the "*Munch*" card game. It consists of a main GUI showing a number of different counters and controls to change their values, together with a menu used for changing configuration preferences (e.g., the maximum counter value). Simply Do (A2) is a ToDo lists management utility. Its main GUI reports the current list of ToDo activities organized in a hierarchy, while many different features to organize, sort, delete, and export a backup of ToDo items are accessible via menus. Tippy Tipper (A3) is a utility app for the calculation of tips for restaurant bills. From its three main GUIs, it allows the user to specify the bill amount, the tipping percentage, and the number of people splitting the bill. It calculates the amount of money that anyone has to pay. All the calculation parameters may be modified by operating on menus. Finally, Trolly (A4), a shopping list manager application, allows users to add/remove objects from lists, and to maintain a list of the most recently listed objects, as well. It manages three kinds of lists, namely, "*items to buy*," "*bought items*," and "*offlist items*" (i.e., items that were in past lists and can be inserted in new lists: This list acts as a suggestion list).

TABLE 1 The Android apps used in our study.

Id	Name	Version	# Classes	# Activities	# Methods	# LOCs	# Exec. LOCs	# Branches
A1	MunchLife[55]	1.4.4	10	2	28	486	184	61
A2	Simply Do56, [56]	0.9.2	46	3	246	3566	1281	419
A3	Tippy Tipper[57]	1.2	42	6	225	2238	999	208
A4	Trolly[58]	1.4	19	2	64	1062	364	120

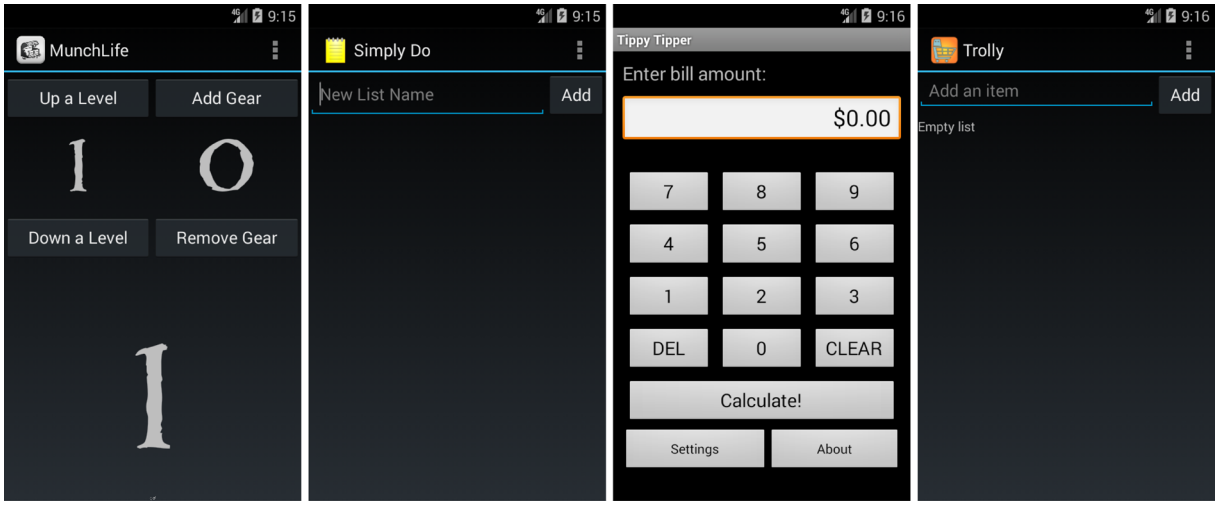

FIGURE 1 Screenshots of the considered AUTs.

The functionalities of these apps are accessible either by app menus or by context menus.

Some of the applications are also sensitive to some system events such as device rotation. A screenshot of the main activity of each of these apps is reported in Figure 1.

Because the selected apps do not interact with any remote service, not even for authentication purposes, GUI testing was possible at system level without the need to deal with precondition setting, mocking, and other integration and system-testing issues.

3.4 | Variables

To compare the effectiveness between the different test suites produced by students, we used the *lines-of-code* (LOC) coverage metric, widely employed in similar works.[53,59] More in detail, we measured LOC coverage using the well-known Emma[49] tool, with respect to the Executable Lines of Code. To compute the coverage achieved by a group of testers, we used Emma to compute the union of the coverages achieved by every single tester.

3.5 | Materials and procedure

In this subsection, we describe the procedure that we adopted for our experiments. Firstly, before the experiments, all the involved students were given several hours of practical lectures on GUI testing techniques and on C&R tools, focusing in particular on Robotium Recorder, to ensure that they all had adequate expertise with the tool.

Subsequently, we assigned the students two testing tasks. The first task consisted of generating a test suite for each of the AUTs using Robotium Recorder and a UET approach. This task was carried out in a controlled environment, in the software engineering laboratory, and all the students were supplied with a virtual machine including the Eclipse IDE, the AUTs, and an instance of an Android emulator capable of running the AUTs. During this task, all students had no prior knowledge of the AUTs, nor they had access to their source code or to code coverage reports. To prevent the need to deal with specific preconditions and to restore the application to a clean state, we suggested the students to produce a single test case for each app, composed of a single, long sequence of interactions. This choice of recommending the recording of a single, longer test case was driven by the desire to simplify the testing process and minimize the effort required to reset the application status and establish pre-conditions for each test sequence, which are not negligible when using C&R tools. Opting for a single, continuous test, allowed us to reduce logistical challenges and save time for the participants. In addition, several empirical studies found that longer test cases had a better fault detection capability in random testing[60] and specification-based testing.[61,62] Still, according to an empirical study presented by Xie and Memon,[63] the length of test case sequences in GUI testing has no significant impact on the fault-detection effectiveness of the resulting test suite, even though it might make fault localization and troubleshooting more challenging. Nonetheless, this approach presents certain trade-offs. For instance, when recording only longer tests, it may be challenging to capture the full range of potential issues, errors, or corner cases that could arise during

real-world usage. Additionally, longer test sequences may introduce higher participant fatigue and reduced attention, which could impact the quality of the tests. Indeed, participants may struggle to maintain focus, leading to potentially suboptimal user interactions.

As an alternative, anyhow, the students could produce different test cases by returning each time to the initial state of the virtual machine. We gave the students a time limit of 4 h to complete this task. Within this time frame, the students were free to decide the order in which to analyze the AUTs, as well as the amount of time to allocate to each AUT.

The second task was a homework assignment in which the students had to improve their previously developed test suites. In this task, each student had to first measure the code coverage of the previously developed test cases, as explained during the lectures. Subsequently, in an IET fashion, based on an analysis of the source code and of the code coverage reports, the students had to generate additional tests using Robotium Recorder and the same testing environment, with the goal of improving code coverage. With the second assignment, performing IET activities starting from the results of the previous UET task, the students implemented a U+IET strategy. The students submitted the output for this task (i.e., the resulting test suites and the corresponding coverage reports) after the end of the Software Engineering course when they were ready to take the final exam. Before the experiments, we made clear to the students that the achieved coverage values were not to be considered in the determination of the course grades, in order to discourage plagiarism among students. On the other hand, the obtained result and the adopted methodologies were objects of discussion in the final examination. Upon submitting their work, students were also asked to report how much time it took them to complete the second task. On average, the students reported that they completed the second task in approximately 8 h. All the tests produced by the students were re-executed by one of the authors to ensure that they worked properly and to validate coverage data.

Subsequently, starting from tests produced by each subject in isolation, we combinatorially computed, for each of the AUTs, for each ET strategy, and for each of the 1,048,575 distinct test subsets arising from our subject set of 20 testers, the aggregate coverage that particular group would have achieved. In Table 2, we report more details on the investigated subsets of testers and show, for each considered cardinality in $[1,\ldots,20]$, the corresponding number of subsets. The number of considered subsets ranges from a single subset of size 20 to 184,756 subsets of size 10. For the sake of compactness, as the number of subsets of cardinality i and $20-i$ is the same for all i in $1,\ldots,20$, we report only one row for both cardinalities. For example, there exist 20 different subsets of a single tester, as well as 20 distinct subsets of 19 testers.

To answer RQ3 and ensure a fair comparison between different-sized groups using different strategies, we took into account the overall effort (measured in man-hours) required to perform each testing task. In particular, in our analysis, we assigned a 1-h effort to uninformed testing tasks, and a 3-h effort to Informed ones (e.g., three testers using an uninformed approach cost the same as one tester performing an Informed approach). We assigned a 3-h effort to U+IET tasks because we consider both the initial uninformed exploration (which lasted 1 h per app) and the subsequent Informed exploration (which lasted on average 2 h per app) as part of the U+IET process.

Furthermore, to answer RQ3, we also computed the average LOC and branch coverage achieved by groups of testers using heterogeneous exploratory strategies (i.e., some testers in the group use a simple UET approach, whereas some others use a U+IET approach). We refer to this heterogeneous approach as *mixed*. Mixed groups can be a very interesting solution for exploratory testing, in scenarios in which it is not possible to share the source code of the AUT with external workers. Indeed, a project manager might recruit a number of external workers using an *uninformed* approach and have some other workers, internal to the company, record test suites with a U+IET strategy. In this case, to face the combinatorial explosion in the number of possible subsets to consider, we limited our analysis to the configurations in which the overall effort in man-hours does not exceed 20 h. We selected this threshold so that the results of *mixed* groups can be compared with both those of UET and U

TABLE 2 Details on the investigated groups of testers.

Subset cardinality	Number of subsets
20	1
1 or 19	20
2 or 18	190
3 or 17	1140
4 or 16	4845
5 or 15	15,504
6 or 14	38,760
7 or 13	77,520
8 or 12	125,970
9 or 11	167,960
10	184,756
Total	**1,048,575**

+IET groups, under the same man-hours budgets. Moreover, for each valid combination of UET and U+IET testers not exceeding the 20-h effort threshold, we randomly sampled 1000 distinct groups, guaranteeing that the subsets of testers using a UET and a U+IET strategy are always disjoint.

The data and code to replicate our findings are publicly available in a replication package.[28] In particular, the package contains:

- the source code of the four Android applications that we considered in our study, including the instrumented versions used to compute branch coverage;
- the test suites developed by the students using UET and U+IET strategies, including code coverage reports;
- Python scripts that we developed to compute the aggregate code coverage of all the considered subsets of students' test suites and of a sample of the subsets including mixes of test suites produced by UET and U+IET tester;
- R scripts that we developed to carry out statistical analyses and draw plots.

4 | RESULTS

4.1 | RQ1: What is the effect on code coverage of adding additional workers to non-communicating groups of testers using a UET (resp., U+IET) strategy?

In Table 3, we report, for each AUT, for each exploratory strategy, and for each considered number of testers, the average LOC and branch coverage percentage achieved by groups of that size.

Figure 2 shows the average and median LOC coverage achieved by different-sized groups of testers, as well as the measured standard deviation. These plots show that, regardless of the considered exploratory strategy, the benefit in terms of test coverage of adding an additional tester to the group is greater the smaller the original group, and becomes negligible as the size of the original group increases.

TABLE 3 Average LOC and branch coverage percentage achieved by different-sized groups of testers.

	MunchLife				SimplyDo				TippyTipper				Trolly			
	UET		U+IET		UET		U+IET		UET		U+IET		UET		U+IET	
Group size	LOC	Br.	LOC	Br.	LOC	Br.	LOC	Br.	LOC	Br.	LOC	Br.	LOC	Br.	LOC	Br.
1	84.1	66.7	93.4	86.6	78.5	67.4	82.8	71.6	83.9	75.2	87.3	76.2	77.5	59.7	83.5	65.1
2	89.7	75.1	95.9	90.9	82.3	73.4	84.0	75.8	86.9	80.2	89.2	82.2	79.0	62.9	84.3	67.3
3	91.7	79.5	96.3	91.8	82.9	74.7	84.6	76.8	87.6	81.5	89.7	83.0	80.0	64.1	85.4	68.2
4	92.8	82.3	96.5	92.1	83.1	75.3	85.0	77.4	88.0	82.1	90.1	83.4	80.5	64.7	86.1	68.6
5	93.5	84.3	96.6	92.2	83.2	75.7	85.2	77.9	88.3	82.5	90.4	83.7	80.8	65.2	86.7	68.9
6	93.9	85.7	96.7	92.3	83.3	76.0	85.4	78.2	88.5	82.8	90.6	83.9	81.0	65.5	87.1	69.0
7	94.3	86.9	96.8	92.4	83.5	76.3	85.6	78.4	88.8	83.1	90.7	84.1	81.1	65.7	87.4	69.1
8	94.6	87.9	96.8	92.5	83.6	76.6	85.7	78.6	89.0	83.3	90.9	84.3	81.2	65.9	87.6	69.1
9	94.8	88.7	96.9	92.5	83.7	76.8	85.8	78.8	89.2	83.6	91.0	84.5	81.3	66.1	87.8	69.2
10	95.0	89.3	97.0	92.6	83.8	77.1	85.8	78.9	89.4	83.8	91.2	84.6	81.4	66.2	87.9	69.2
11	95.2	89.8	97.1	92.7	83.9	77.3	85.9	79.0	89.6	84.0	91.3	84.8	81.4	66.3	88.1	69.2
12	95.4	90.3	97.2	92.8	84.0	77.5	86.0	79.1	89.8	84.2	91.4	84.9	81.4	66.4	88.1	69.2
13	95.5	90.6	97.3	92.9	84.1	77.6	86.0	79.1	90.0	84.4	91.5	85.0	81.4	66.5	88.2	69.2
14	95.7	90.9	97.3	93.0	84.3	77.8	86.0	79.2	90.2	84.6	91.6	85.1	81.4	66.5	88.2	69.2
15	95.8	91.2	97.4	93.0	84.4	77.9	86.1	79.2	90.4	84.7	91.6	85.2	81.4	66.6	88.3	69.2
16	95.9	91.3	97.5	93.1	84.5	78.1	86.1	79.3	90.5	84.9	91.7	85.3	81.4	66.6	88.3	69.2
17	96.0	91.5	97.6	93.2	84.6	78.2	86.2	79.3	90.7	85.1	91.8	85.4	81.4	66.6	88.3	69.2
18	96.1	91.6	97.7	93.3	84.7	78.3	86.2	79.4	90.9	85.2	91.8	85.5	81.4	66.7	88.3	69.2
19	96.1	91.7	97.7	93.4	84.8	78.4	86.2	79.4	91.1	85.4	91.9	85.5	81.4	66.7	88.3	69.2
20	96.2	91.8	97.8	93.4	84.9	78.5	86.3	79.5	91.2	85.6	91.9	85.6	82.1	66.7	88.9	69.2

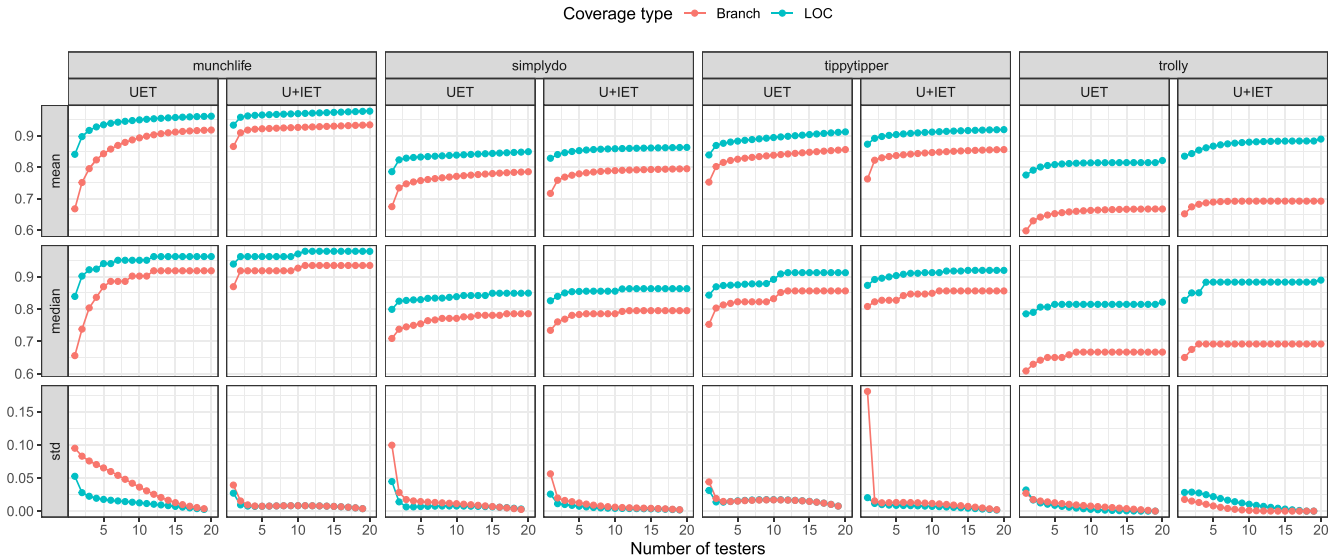

FIGURE 2 Average, median, and st. dev. of the LOC and branch coverage achieved by different-sized groups of testers.

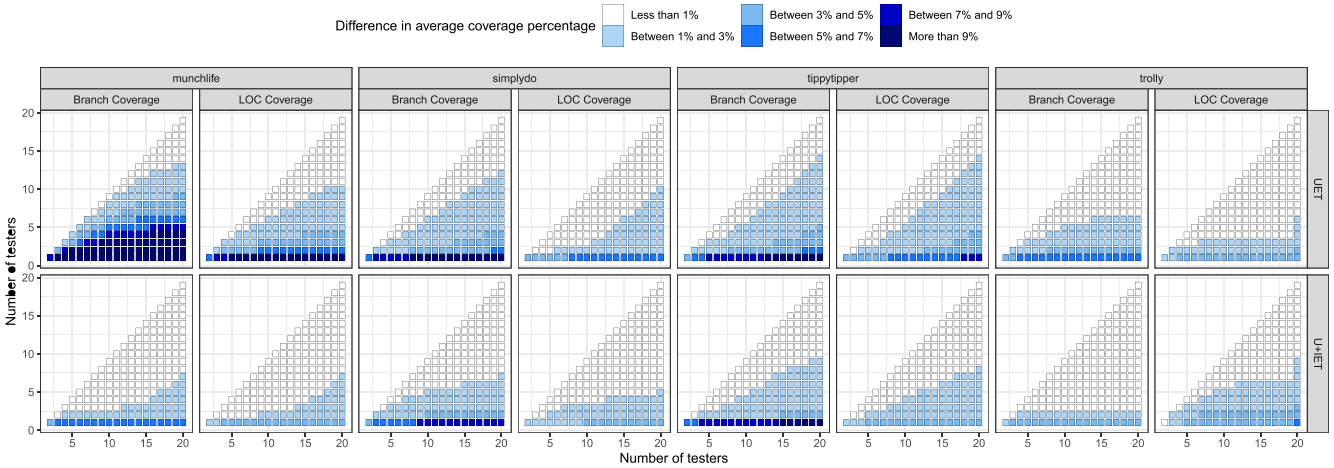

FIGURE 3 Difference in LOC and branch coverage percentage achieved by different-sized groups of testers using the same exploratory testing strategy.

To better visualize the differences in terms of achieved LOC and branch coverage between different-sized groups of testers using the same exploratory strategy, in Figure 3, we depict such differences using tile diagrams. In that figure, darker tiles are associated with higher differences in coverage.

We can observe that the area with white tiles (and, in general, lighter tiles) is more extended in the diagrams corresponding to groups of U+IET testers, independently from the AUT. In other words, the benefit on LOC and branch coverage of adding an additional tester to the group is generally greater for UET testers with respect to U+IET testers.

Moreover, the results also highlight that the introduction of additional testers often leads to greater returns in terms of branch coverage w.r.t. LOC coverage, especially when testers use a UET strategy. This is witnessed by the generally darker areas at the bottom of each tile plot representing the differences in terms of branch coverage.

This phenomenon is explained by the diverse perspectives and varied approaches brought by each individual tester. Indeed, when multiple testers are involved, they are likely to explore the application's GUI from different angles, generating a broader range of test scenarios exercising different user flows, input combinations, and error conditions, leading to improved coverage of the different decision points and branches in the codebase. Such an increase in branch coverage does not necessarily imply a similar increase in LOC coverage due to the fact that branch coverage focuses on examining different decision points and possible outcomes in the code, and does not account for every individual line of code within

those branches. Indeed, it is possible to increase branch coverage without increasing LOC coverage. For example, covering a path not entering an `if` statement leads to an increment in branch coverage while the LOC coverage remains the same.

To determine whether the differences in LOC and branch coverage percentage among different-sized groups are statistically significant, we performed statistical tests. In particular, for each AUT and exploratory strategy s, and for each pair of cardinalities (k,k'), with $k,k' \in [1,...,19], k > k'$, and for each considered code coverage type (i.e., branch and LOC coverage), we tested the null hypothesis: $\mathbf{H_0^{AUT,s,k,k'}}$: The code coverage achieved using the exploratory strategy s on AUT by k testers is smaller than or equal to the one achieved by k' testers.

Notice that we do not include in our statistical analysis groups of cardinality 20 because, as shown in Table 2, only one subset of that cardinality exists in our setting, and thus, the reduced sample size hinders the applicability of statistical tests.

To test the null hypotheses, we first tested the LOC and branch coverage distributions for normality using the Shapiro–Wilk test. Then, when both the distributions of code coverage were normal, we used the Student's t-test to evaluate the null hypotheses. In the other cases, we used the Mann–Whitney–Wilcoxon test,[64] which does not assume normal distributions. When the test p-value < 0.05, we reject the null hypothesis with high confidence, accepting the alternative hypothesis that the code coverage achieved by k testers is greater than that achieved by k' testers. Statistical test results are reported with tile plots in Figure 4.

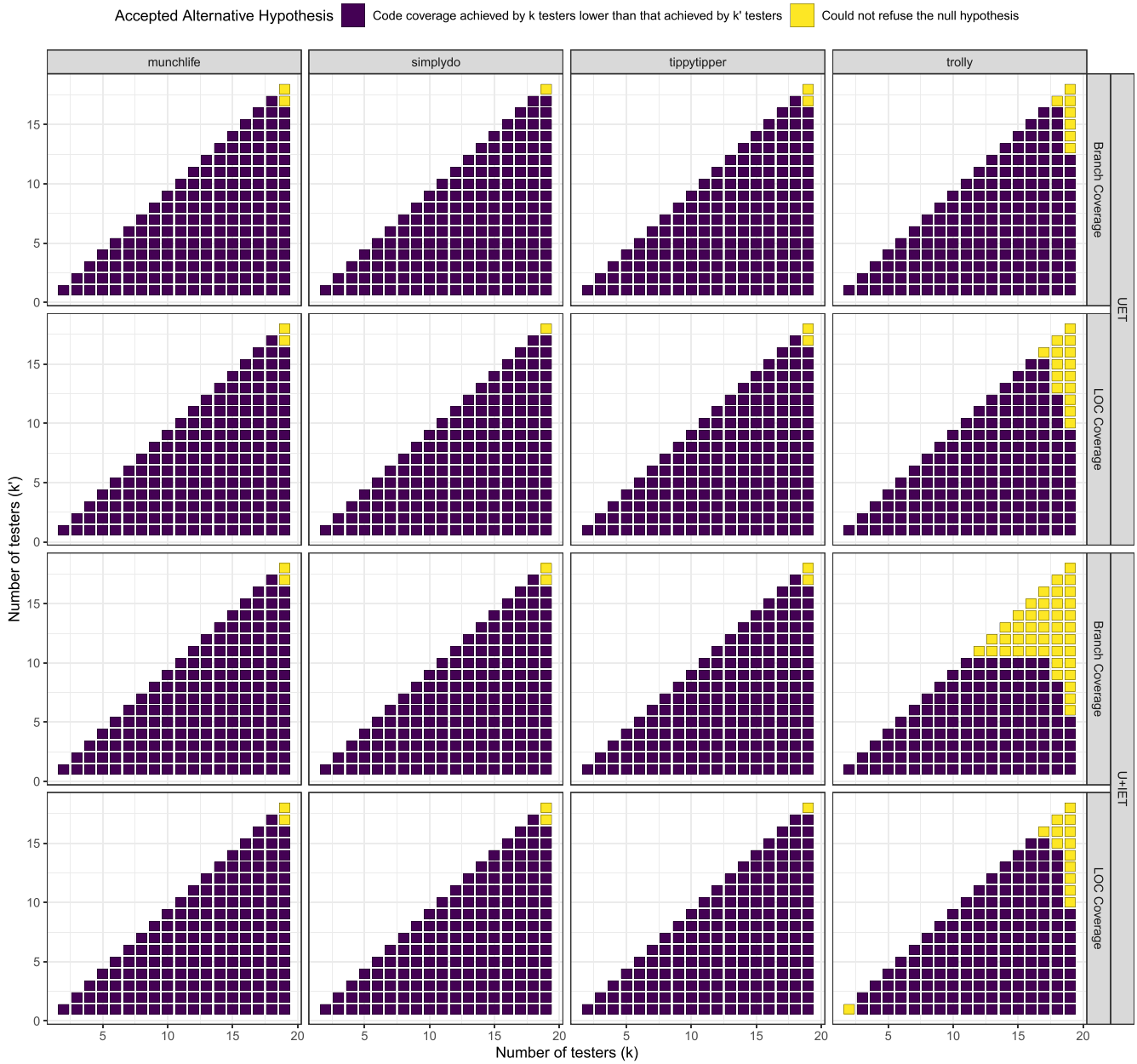

FIGURE 4 Results of hypotheses testing: comparing the LOC and branch coverage of test suites generated by different-sized groups using the same exploratory strategy.

In the figure, light tiles correspond to null hypotheses that we could not reject, whereas dark tiles correspond to scenarios in which the tests confirmed statistically significant differences in code coverage. More in detail, the dark tiles are associated with scenarios in which the number of testers on the *x*-axis achieves a greater coverage than the corresponding number of testers on the *y*-axis. As the plot highlights, in almost every scenario, we detected statistically significant improvements in both LOC and branch coverage when increasing the number of testers, with the only exceptions being situations in which the original group had already a large number of testers (light area in the upper-right corner of each tile plot in Figure 4).

Even when statistically significant, however, the difference in LOC and branch coverage achieved by different-sized groups of testers might be negligible and have no practical impact. Hence, we measured the magnitude of these differences using the *Cliff's delta* effect size,[65] a metric that is largely used in software engineering studies to compare the degree of difference between the two experimental results.[66] Cliff's delta ranges between −1 and 1, and can be interpreted as follows: If $|\delta| < 0.147$, the difference is *negligible*; if $0.147 \leq |\delta| < 0.33$, the difference is *small*; if $0.33 \leq |\delta| < 0.474$, the difference is *medium*; if $|\delta| \geq 0.474$ the difference is considered *large*. Results of this analysis are shown in Figure 5.

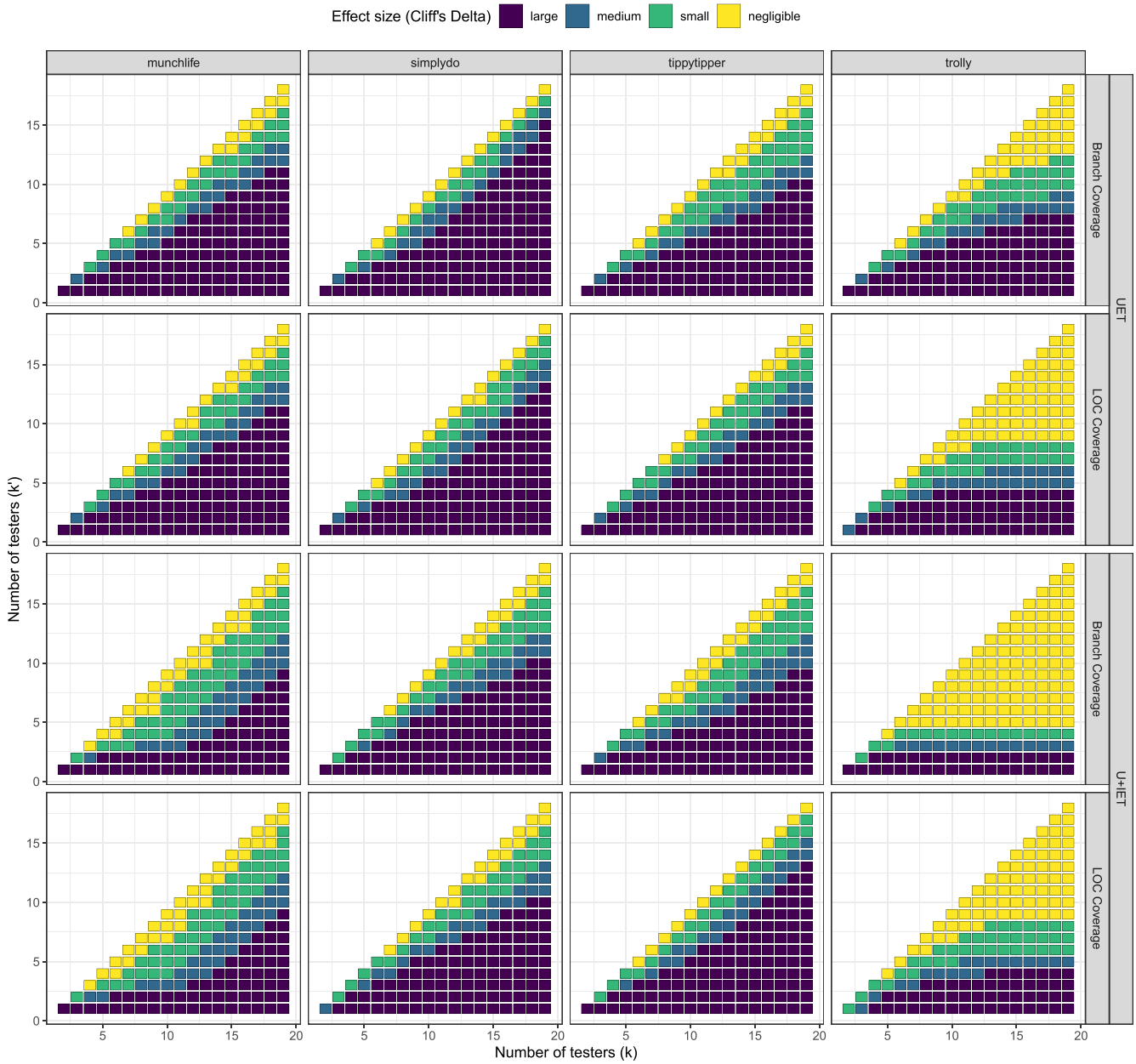

FIGURE 5 Measured effect size (Cliff's Delta): comparing the LOC and branch coverage of test suites generated by different-sized groups using the same exploratory strategy.

RQ1: Results highlight that, regardless of the considered exploratory strategy, increasing the size of the group of testers by small amounts (e.g., 1 to 3 additional testers) generally leads to small or negligible improvements in both LOC and branch coverage, especially when starting with an already consistent group (e.g., more than 10 testers).

Figures 2 and 3 show that the greater sensitivity to the increase in the number of UET testers is observable for MunchLife, for which there are steeper slopes of the curves related to the average and to the median of the LOC coverage (see Figure 2). Similarly, the darker area is larger for MunchLife (see Figure 3). We analyzed the source code and the coverage reports produced by the UET testers for this application, and we noticed the presence of relatively large areas of code that were executable only by test cases including device orientation change events. The presence of device orientation-dependent code is unusual in small-sized Android apps and most UET testers did not consider these events in their test cases because their systematic inclusion certainly would have remarkably increased the time needed to capture the test cases. In addition, other chunks of code that have not been covered by many UET testers are related to complex sequences of events, not self–explained into the GUI screens (they were described in greater detail by Di Martino et al.[8]).

In conclusion, the lack of knowledge about the application under test and the strict time limit represent the main causes for the variability of the coverage results achieved by different UET testers.

On the other hand, residual differences in terms of coverage achieved can be observed for U+IET testers, too. For example, some complex interactions in SimplyDo and TippyTipper have been found by a few U+IET testers. In addition, a specific undocumented and unintuitive feature of Trolly (i.e., the "offlist" list) represented an unsolvable coverage problem for about half of the U+IET testers, and it is responsible for the few darker tiles observable in Figure 3 for Trolly and U+IET testers.

In general, it appears that the residual differences in coverage between different U+IET testers depend on their ability to produce effective test cases: Although they received a common background, some significant differences between their results can be observed.

4.2 | RQ2: How do the test suites generated by different-sized groups of testers with a UET strategy compare, in terms of code coverage, to those generated by different-sized groups of testers with a U+IET strategy?

In order to answer this research question, we investigate the differences in coverage achieved by different-sized groups of testers using different exploratory strategies. In particular, for each of the AUTs and for each combination of the number of testers $(k,k'), k,k' \in [1,...,20]$, we computed the difference between the LOC and branch coverage achieved on average by a group of k non-communicating testers using a U+IET strategy and the one achieved by k' testers using a UET strategy. Negative values correspond to cases in which UET testers achieved a higher LOC and branch coverage, whereas positive values, on the other hand, correspond to scenarios in which U+IET testers achieved a higher code coverage. In Figure 6, we report the results of this analysis.

The white areas in the figure correspond to cases in which the code coverage achieved using UET and U+IET strategies is comparable (less than 1% difference in absolute value), red areas correspond to situations in which the coverage achieved by UET testers is greater than that achieved by U+IET testers, while blue areas correspond to situations in which U+IET testers achieved higher coverage. As Figure 6 highlights, when considering LOC coverage, in three of the four considered AUTs, namely, MunchLife, SimplyDo, and TippyTipper, a noticeable equivalence area exists, and in some cases, the LOC coverage achieved by larger groups of UET testers (i.e., generally more than 10 testers) is higher than that achieved by smaller groups of U+IET testers (i.e., 1 to 3 testers). In Trolly, on the other hand, the LOC coverage achieved even by a single U+IET tester exceeded that of a group of 20 UET testers by more than 1%, and therefore, no equivalence area exists. When considering branch coverage, a similar pattern occurs in the first three AUTs, with slightly larger differences in coverage percentages, witnessed by the generally larger darker areas. In Trolly, on the other hand, we can observe that larger numbers (i.e., > 10) of UET testers achieve a slightly higher branch coverage than a single U+IET tester. The fact that a single U+IET tester achieved on average a better LOC coverage than any number of UET testers is explained, as already observed when analyzing the results for RQ1, by the diverse perspectives and varied approaches brought by each individual UET tester, and by the fact that U+IET testers were driven in their testing process by the maximization of LOC coverage rather than branch coverage. Indeed, the UET testers might have covered more branches by simply exercising different input combinations, driven by their personal sensibility, while the U+IET testers had no incentive to cover more branches if this led to no return on LOC coverage.

To determine whether these differences in LOC and branch coverage are statistically significant, we performed statistical tests and measured the effect size following the same procedure described in the previous subsection. More in detail, for each AUT and for each pair of cardinalities (k,k'), with $k,k' \in [1,...,19]$, and for each considered code coverage type (i.e., LOC and branch), we tested the null hypotheses: $\mathbf{H}_{0-\text{smaller}}^{\text{AUT},k,k'}$: The code coverage achieved using a UET strategy on AUT by k testers is smaller than or equal to the one achieved by k' testers using a U+IET strategy.

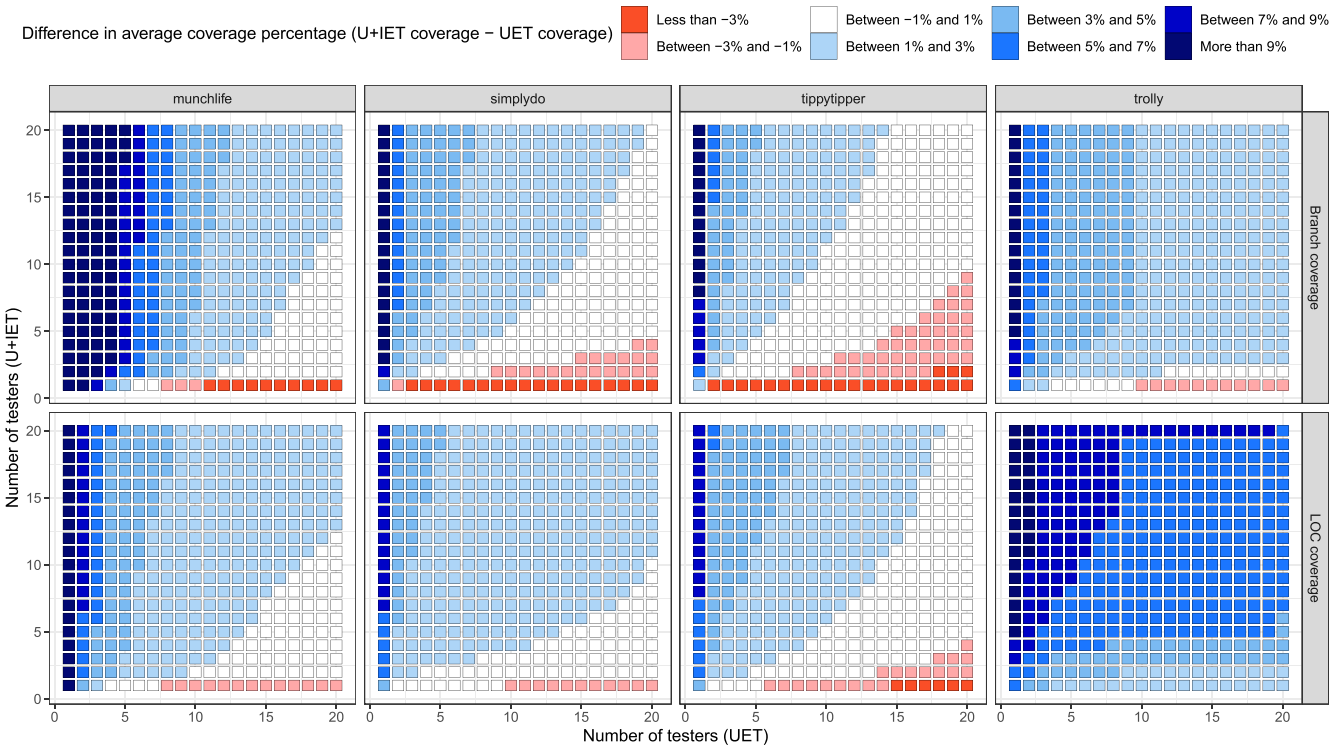

FIGURE 6 Difference in LOC and branch coverage achieved by different numbers of testers using different testing strategies.

$\mathbf{H}_{0-\text{greater}}^{\text{AUT},k,k'}$: The code coverage achieved using a UET strategy on AUT by k testers is greater than or equal to the one achieved by k' testers using a U+IET strategy.

To test the null hypotheses, we used the Student's t-test when both the sample distributions are normal, and the Mann–Whitney–Wilcoxon test[64] otherwise. When the test p-value < 0.05, we reject the null hypothesis with high confidence, accepting the alternative hypothesis that the LOC and branch coverage achieved by k testers using UET is greater (resp., smaller) than that achieved by k' testers using U+IET. Statistical test results are reported with a tile plot in Figure 7.

In the figure, blue tiles correspond to cases in which we could reject $\mathbf{H}_{0-\text{greater}}^{\text{AUT},k,k'}$, thus accepting the alternative hypothesis that the LOC and branch coverage achieved by k testers using UET is smaller than that achieved by k' testers using U+IET. Red tiles, on the other hand, correspond to cases in which we could reject $\mathbf{H}_{0-\text{smaller}}^{\text{AUT},k,k'}$, thus accepting the alternative hypothesis that the LOC and branch coverage achieved by k testers using UET is greater than that achieved by k' testers using U+IET. Lastly, white tiles correspond to cases in which we could not reject any null hypotheses, and thus, we can draw no statistical conclusion. Results show that, in a preponderant number of cases, the code coverage achieved by groups of testers using UET is smaller, in a statistically significant way, than that achieved by groups of U+IET testers.

In MunchLife and SimplyDo, larger groups of UET testers manage to achieve a statistically higher LOC and branch coverage than smaller groups of U+IET testers (size smaller than 5). In particular, in MunchLife, sets of 7 or more UET testers achieve a higher LOC and branch coverage than sets with a single U+IET tester, and it takes 16 or more UET testers to achieve a higher LOC and branch coverage than a pair of U+IET testers. In SimplyDo, it takes 4 or more UET testers to cover more LOC than a single U+IET tester, while 14 or more UET testers are needed to achieve better LOC coverage than a pair of U+IET testers. When considering branch coverage, 3 or more UET testers are required to cover more branches than a single U+IET tester, and more than 5 U+IET testers are necessary to achieve an equivalent or better branch coverage than any number of UET testers. In both MunchLife and SimplyDo, even 20 UET testers do not achieve a higher LOC or branch coverage than 6 or more U+IET testers. In TippyTipper, there is a larger number of cases in which UET groups achieve a higher code coverage than U+IET groups, as witnessed by the larger red area in the bottom-right corner of the tile diagrams. Nonetheless, 9 or more U+IET testers achieve a higher or comparable LOC coverage than any number of UET testers. When considering branch coverage, it takes 17 or more U+IET testers to achieve a code coverage comparable with 20 UET testers. Lastly, in Trolly, a single U+IET tester achieves a higher LOC coverage than any number of UET testers, and two U+IET testers achieve a better branch coverage than any number of UET testers.

The reason behind the better results in terms of LOC coverage obtained by U+IET testers compared with large groups of UET testers in Trolly is related to the existence of the "hidden" functionality related to the management of "offlist" items, which can only be activated with long click events in a particular execution scenario: It was discovered almost exclusively by the U+IET testers who could observe the source code

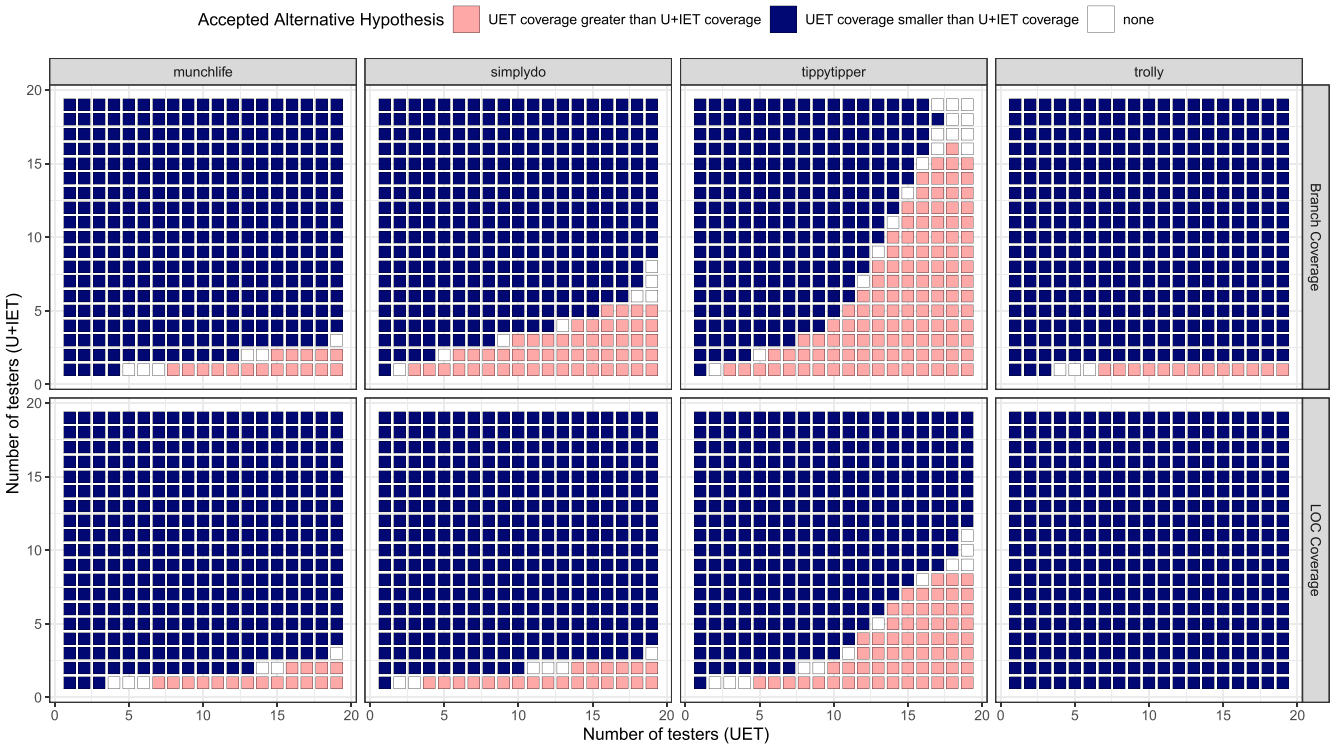

FIGURE 7 Results of alternative hypotheses testing: comparing the LOC and branch coverage of test suites generated by different-sized groups using different exploratory strategies.

related to its implementation. Still, covering this hidden functionality leads to a significant increase in LOC coverage but has a limited impact on branch coverage, which explains the branch coverage trend.

On the contrary, the better results in terms of both LOC and branch coverage obtained by large sets of UET testers compared with small sets composed of one or more U+IET testers observed for TippyTipper depend on the absence of parts of the AUT that are difficult or intuitive to reach and test. In these cases, UET testers were limited mainly by the strict time limit and their diversity allowed them to obtain excellent results by considering the union of the coverage obtained by a large group of them, especially in terms of branch coverage, as already observed also in the response to RQ1.

Even when statistically significant, however, the differences in LOC and branch coverage achieved by different-sized groups of testers using different exploratory strategies might be negligible. Hence, we measured the magnitude of these differences using the *Cliff's delta* effect size, as discussed in Section 4.1. Results of this analysis are shown in Figure 8 and highlight that in three of the considered AUTs, namely, MunchLife, SimplyDo, and TippyTipper, there exists an "*equivalence*" zone in which different numbers of UET and U+IET testers achieve comparable coverage (i.e., the effect size is small or negligible). In Trolly, on the other hand, the LOC coverage achieved by U+IET testers is always greater than that achieved by any number of UET testers, with a large or medium effect size. When considering branch coverage, a single U+IET tester achieved comparable (effect size small or negligible) branch coverage w.r.t. 4-7 UET testers, but 8 or more UET testers achieved a significantly higher branch coverage with a medium to large effect size. 3 or more U+IET testers achieved a significantly higher branch coverage with a larger effect size than any number of UET testers.

The performance of both U+IET and UET groups and the extent of such equivalence areas are significantly influenced by the nature of the AUT. In AUTs in which some features are unintuitive, *Informed* approaches prove to be significantly more effective, as testers using an *uninformed* strategy often fail to stress these '*hidden*' features. In AUTs featuring a larger number of functionalities, on the other hand, the coverage achieved by each tester appears to be mainly limited by the given time budget. In these cases, leveraging larger numbers of testers using an *uninformed* approach leads to overall better coverage.

RQ2: Results highlight that small groups of U+IET testers generally achieve similar code coverage results w.r.t. larger groups of UET testers. In apps with unintuitive features, even a single U+IET tester can achieve better results than a large group of UET testers.

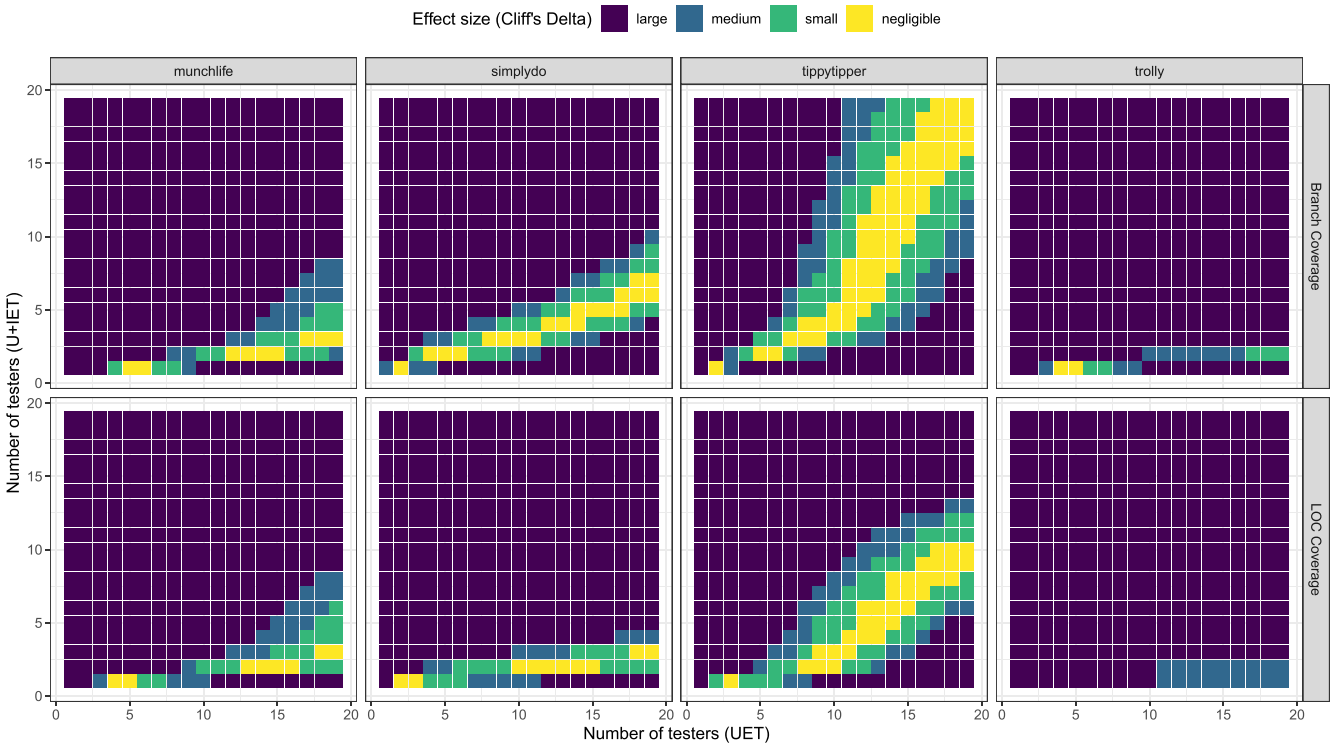

FIGURE 8 Measured effect size (Cliff's Delta): comparing the LOC and branch coverage of test suites generated by different-sized groups using different exploratory strategies.

4.3 | RQ3: Under a fixed budget, how do test suites generated by testers using a UET, U+IET, and mixed approach compare, in terms of code coverage?

Generating tests with a U+IET strategy includes a preliminary *uninformed* exploration, and a subsequent *Informed* exploration phase, driven by an analysis of code coverage reports. As a consequence, U+IET is more time-consuming than a simpler *uninformed* exploratory approach. The goal of this research question is to investigate which exploratory strategy (or mix thereof) is the most cost-effective. As discussed in Section 3.5, we assigned a 1-h effort to *uninformed* testing tasks, and a 3-h effort to the U+IET ones.

In Table 4, we report the average LOC coverage percentage achieved by different-sized groups of testers using different strategies, with the same overall man-hours effort. In particular, we consider (I) groups using an *uninformed* approach; (II) groups using an *Informed* approach; and (III) groups in which some testers use a U+IET strategy, while some others use a UET approach (which we refer to as *mixed*). For each AUT and for each overall man-hours budget, we highlight in bold the maximum LOC and branch coverage achieved by any of the considered strategies. For the sake of compactness, we report in Table 4 only a subset of the overall man-hours budgets for which a U+IET group exists. Complete results are available in the replication package.[28]

Results show that, under the same effort budget, the LOC coverage achieved by U+IET groups is always greater than or comparable (less than 0,5% difference) to the one achieved by UET groups. In some cases, the difference in LOC coverage between U+IET and UET groups is more noticeable. In MunchLife, for example, with a 6-h overall effort, 2 U+IET testers achieve on average 2% more LOC coverage than 6 UET testers. Similarly, in Trolly, with an 18-h budget, 6 U+IET testers achieve 5,7% more LOC coverage than 18 UET testers. In most cases, however, the difference is less noticeable (1% or less). When considering branch coverage, especially in scenarios with lower effort budgets and on larger applications such as SimplyDo and TippyTipper, larger groups of UET testers manage to achieve a higher code coverage than smaller groups of U+IET testers. For example, in the 3-h effort budget scenario, 3 UET testers achieve a 3,1% (resp., 5,3%) higher branch coverage than a single U+IET tester on SimplyDo (resp., TippyTipper). As for mixed groups, we can observe that, for small budgets (e.g. 6, 9 or 12 man-hours), the coverage percentage achieved with mixed approaches has intermediate values w.r.t. UET and U+IET testers. In these cases, the effectiveness of the resulting mixed test suites appears to be generally greater than that of UET groups, and comparable with that of U+IET groups. On the contrary, as the budget increases to 15 or 18 man-hours, there are frequent cases in which combinations including both UET testers and U+IET testers provide the best code coverage results. For example, with an 18-h overall effort, a mixed group of 5 U+IET testers and 3 UET testers achieves higher LOC and branch coverage than a U+IET group of 6 testers on both MunchLife. In both cases, however, the difference in LOC coverage is rather small (0,2%).

TABLE 4 LOC and branch coverage percentage achieved using different strategies, with the same overall man-hours effort.

Overall effort (man-hours)	Strat.	Number of U+IET testers	Number of UET testers	MunchLife		SimplyDo		TippyTipper		Trolly	
				LOC	Br.	LOC	Br.	LOC	Br.	LOC	Br.
3	UET	0	3	91.7	79.5	82.9	**7**4.7	**87**.6	**81**.5	80.0	64.1
	U+IET	1	0	93.4	86.6	82.8	71.6	87.3	76.2	83.5	65.1
6	UET	0	6	93.9	85.7	83.3	76.0	88.5	82.8	81.0	65.5
	U+IET	2	0	95.9	90.9	84.0	75.8	89.2	82.2	84.3	67.3
	Mixed	1	3	95.6	89.9	83.7	76.0	88.9	82.8	83.0	66.6
9	UET	0	9	94.8	88.7	83.7	76.8	89.2	83.6	81.3	66.1
	U+IET	3	0	96.3	91.8	84.6	76.8	89.7	83.0	85.4	68.2
	Mixed	1	6	95.9	90.8	84.0	76.9	**89**.8	83.6	83.1	67.0
	Mixed	2	3	96.2	91.4	84.3	77.0	89.6	82.3	85.6	68.2
12	UET	0	12	95.4	90.3	84.0	77.5	89.8	84.2	81.4	66.4
	U+IET	4	0	96.5	92.1	85.0	77.4	90.1	83.4	86.1	**68**.6
	Mixed	1	9	96.1	91.2	84.3	77.2	89.9	83.9	83.2	67.2
	Mixed	2	6	96.3	91.7	84.5	76.7	90.4	83.8	85.1	68.2
	Mixed	3	3	96.2	92.1	84.1	77.4	90.0	83.8	84.7	68.2
15	UET	0	15	95.8	91.2	84.4	77.9	90.4	84.7	81.4	66.6
	U+IET	5	0	96.6	92.2	85.2	77.9	90.4	83.7	86.7	68.9
	Mixed	1	12	96.2	91.5	84.6	77.7	90.4	84.5	83.2	67.4
	Mixed	2	9	96.1	91.8	84.5	77.8	90.4	84.5	84.4	67.8
	Mixed	3	6	96.8	91.9	85.1	77.8	90.6	83.8	86.1	68.9
	Mixed	4	3	96.3	91.9	85.2	77.9	90.4	83.9	85.4	68.4
18	UET	0	18	96.1	91.6	84.7	78.3	90.9	85.2	81.4	66.7
	U+IET	6	0	96.7	92.3	85.4	78.2	90.6	83.9	**87**.1	69.0
	Mixed	1	15	96.2	91.7	84.8	78.2	90.8	85.1	83.2	67.5
	Mixed	2	12	96.4	91.9	85.0	78.0	90.7	84.6	84.0	68.0
	Mixed	3	9	96.4	92.1	85.1	77.9	90.6	84.5	85.4	68.6
	Mixed	4	6	96.6	92.3	85.3	78.0	90.7	84.2	85.2	68.7
	Mixed	5	3	96.8	92.5	85.4	78.1	90.6	83.6	86.4	68.9

These results support the insight on the existence of a trade-off between the advantages of the U+IET approach (which leads to test suites achieving significant coverage even with a few testers but requires more time) and those of the UET approach (which can benefit to a greater extent from increasing the number of testers, due to the greater diversity among the results of individual testers, especially in terms of branch coverage). This trade-off could be optimized by a mixed approach involving both UET and U+IET testers.

RQ3: Results highlight that, in the considered setting, using a U+IET strategy is generally more cost-effective than using an *uninformed* approach or combinations of both approaches. Moreover, mixed groups appear to be generally more cost-effective than UET ones and can represent a valid trade-off solution when larger budgets are available, or when there exist particular constraints on the distribution of source code (e.g., it is not possible to share source code with external workers).

5 | THREATS TO VALIDITY

In this subsection, we discuss some threats that could have affected the results of the experiments and their generalization, according to the guidelines proposed in Wohlin et al..[67]

5.1 | Threats to internal validity

To ensure a fair comparison between the effectiveness of the uninformed exploratory testing (UET) task with the one of the U+IET task, we enforced the same experimental conditions for all the experiments. To this aim, each student acted on freshly installed apps, without any data from previous executions, in the context of an instance of the same Android emulator with the same Java virtual machine. In addition, we provided students with a pre-installed instance of the Eclipse IDE, integrated with Robotium Recorder and Android support, and with projects corresponding to the applications under test. In this way, we mitigated the threat related to the waste of testing time due to environment setup activities that could differ between different students. All the students had a preventive knowledge of that environment that was presented in previous lectures of the same course. Furthermore, we selected apps with no dependencies on external resources such as remote data sources or services, so that their behavior was not influenced by external factors. We did not control the order of analysis of the apps, having required that the students freely organized and accomplished their tasks. Of course, it is not possible to exclude the threat of a learning effect, in case all the students analyzed the apps exactly in the same order. In addition, we were not able to reconstruct the time allocated by any student to the testing of each specific application because we asked only to provide the total time spent on the testing of all the apps, without tracing the ending time for each application testing.

5.2 | Threats to construct validity

These threats concern the appropriateness of the measures used for capturing the dependent variables. We measured testing effectiveness in terms of code coverage percentage both at LOC and branch granularity levels. Code coverage is a test adequacy indicator largely used in the literature[68] and can be automatically and accurately measured by automatic tools, such as the widely-used Emma coverage tool. Still, code coverage alone may not capture the full extent of the test's effectiveness in terms of identifying faults or detecting functional or non-functional issues. To address this threat, future replications of the experiments should involve also real or seeded faults, and consider the actual fault-detection capability.

The UET testing activities were performed in a laboratory, with a maximum time of 4 h, which was respected by all students. As for U+IET activities, instead, students were asked to report the testing time they actually dedicated to this activity at home. To mitigate the possibility of students providing inaccurate evaluations of their testing time (in particular underestimated values), it was explained to them that the quality judgment about their activities would not be affected by the declared testing time. In order to compare the effectiveness of different-sized groups using different strategies under the same budget and to provide an answer to RQ3, we have approximated the average time spent by each UET (resp., U+IET) tester on each app to 1 h (resp., 3 h). This approximation could lead to inaccurate results when performing comparisons under the same budget.

5.3 | Threats to conclusion validity

The limited number of experimenters (20 students) and apps (4 small-sized apps) limits the reliability of statistical inferences. In particular, the null hypotheses supporting the research questions RQ1 and RQ2 are based on sets of samples having very different sizes (as shown in Table 2). In order to mitigate this threat, Cliff's delta effect size measure was also taken into account to improve confidence in the conclusions related to the rejection of null hypotheses.

5.4 | Threats to external validity

These threats limit the generalizability of the results and are often posed by the way experimental objects and subjects are selected. In this study, we considered four simple, small-sized open-source Android apps, selected according to the inclusion criteria listed in Section 3 and used in previous studies on Android app testing.

Such a small sample may not accurately represent the wide range of applications available, thus limiting the generalizability of the findings. Additionally, small-sized applications may differ from larger and more complex ones, raising concerns about the applicability of the results to broader contexts. The limited number of applications also reduces the variability in features, user interface widgets, and underlying code structures, posing additional challenges to the generalizability of the findings. Furthermore, the selected open-source applications may not fully reflect industry-standard practices and may not be representative of applications developed by larger teams, which could restrict the external validity in commercial contexts. Additional experiments involving a larger sample size and incorporating a more diverse set of applications, possibly including commercial projects, should be carried out in order to mitigate these threats.

As for the C&R tool, in our experiments, we used the Robotium Recorder tool. Currently, Android Espresso Test Recorder, a free tool included in the Android Studio suite, has surpassed Robotium Recorder in popularity. Anyhow, the features of the two tools are quite similar. We performed other experiments with different graduate students using Android Espresso Test Recorder in a UET scenario on the same AUTs, and the results showed that there are no significant differences in the effectiveness of the recorded test cases with respect to the ones recorded with Robotium.

Concerning the selected subjects, the involvement of students as participants may affect the generalization of results.[69,70] Anyway, different recent studies state that it is reasonable to recruit testers from the researchers' organization.[26,27] The 20 students involved in this experiment had little experience in Android development and GUI testing, and several studies[71–73] observed that graduate students and practitioners perform similarly when applying a new technology during experimentation. Thus, we suggest that our results could be generalizable to exploratory testing scenarios in which C&R testers with limited testing skills are recruited.[37]

An additional threat to the generalizability of our findings is posed by the fact that the subjects of our study were suggested to record a single, longer GUI test instead of implementing multiple, distinct tests. This choice was driven by the desire to simplify the testing process and minimize the effort required to reset the application status and establish pre-conditions before each test sequence, to reduce logistical challenges and save time for the participants. Designing longer test sequences has been shown to have no significant impact on the fault-revealing capability of test suites,[63] but, nonetheless, this approach might not be representative of industrial practices, which generally favor having multiple GUI tests with smaller sequences of user interactions, to ease the identification of specific issues and enable targeted troubleshooting.

6 | CONCLUSIONS

In GUI testing of mobile applications, exploratory testing strategies are often used when designing tests. These strategies emphasize the individual tester's autonomy and accountability, encouraging them to continuously enhance the quality of their work, treating test design, execution, and result interpretation as concurrent interconnected activities. While a number of works have focused on investigating the effectiveness of exploratory testing, effectively managing its application and allocating an adequate amount of resources remains an open challenge, due to the relative unpredictability of exploratory testing activities, which are greatly influenced by the skills and abilities of individual testers.

In this work, we have investigated how the effectiveness of the test suites produced by groups of independent testers may vary depending on the size of the group and the type of exploratory testing strategy they exploit. To this aim, we have elaborated on the results of a previously published empirical study[8] involving 20 master's students, who can be considered representatives of practitioners taking part in exploratory testing activities. The students were assigned two tasks aiming at the production of effective test suites for a set of four small-sized Android apps, using a C&R tool and different exploratory strategies. In particular, in the first task, they generated tests by using an *uninformed* exploratory strategy of the apps (without any information about the source code of the apps and the achieved code coverage), whereas in the second task, they used an *Informed* exploratory strategy, that is, they took into account the knowledge of the source code and of the code coverage achieved by the previously recorded test cases.

The analysis of the experimental results provided some interesting conclusions and insights. First of all, we have observed that increasing the size of the group of allocated testers by small amounts generally leads to small or negligible improvements in coverage, especially for larger groups. The comparisons between the coverage results achieved by groups of UET and U+IET testers showed that small groups of U+IET testers generally achieve similar results with respect to larger groups of UET testers, but the test suites produced by U+IET testers are superior with respect to the coverage of complex or unintuitive functionalities. The comparisons between the results of different groups having the same overall testing effort (in terms of man-hours) and composed of UET testers, U+IET testers or a mix of them, provided some insights about the optimal composition of groups. In the context of our experimental setting, U+IET groups were generally more cost-effective than UET groups or mixed groups, whereas the latter generally outperform UET groups, and appear as a viable alternative when larger budgets are available.

Although our study provides some first insights into the code coverage achievable by different-sized groups of testers using exploratory approaches, future research should focus on improving the generalizability of these findings by conducting additional experiments involving more complex, industrial-strength Android applications. Moreover, it would be valuable to investigate the application of exploratory testing in a crowdtesting context, leveraging larger groups of crowdtesters recruited using online services in place of students, and considering real crowdtesting costs. Furthermore, future investigation on more complex applications could also consider additional metrics to capture test effectiveness, such as fault-detection capability (with real or seeded faults), or functional coverage criteria.

With additional experiments in place, future works could also explore the feasibility of defining a regression model capable of estimating the code coverage achievable by a crowd of testers of a certain size on a given AUT, considering some characteristics (e.g., number of LOCs, number of branches, number of activities, etc.) of the application as features. Such a model could have a significant practical impact on the testing process, by supporting project managers in allocating the "right" amount of crowdtesting resources to test their applications.

An additional extension direction that might be worth investigating involves studying the effect of the skill level of the recruited crowdtesters on the effectiveness of the resulting test suites. This could be done by conducting additional experiments involving both Bachelor

of Science (BSC) and Master of Science (MSC) students. By comparing the code coverage achieved by crowds of BSC and MSC students, it would be possible to investigate the degree to which more advanced academic training and specialization could influence test effectiveness. Such investigation holds the potential to provide valuable insights into the role of skill level in test performance and can offer practical implications to project managers recruiting resources for exploratory crowdtesting. Furthermore, future works could also be directed toward designing similar experiments in the different domain of End-to-End testing of web applications, to investigate the extent to which our findings could translate to that domain.

ACKNOWLEDGMENTS

This work was partially funded by the PNRR MUR Project FAIR (Future Artificial Intelligence Research) under the number PE0000013 and by the ENACTEST Project (European iNnovation AllianCe for TESting educaTion)- ERASMUS-EDU-2021-PI-ALL-INNO under the number 101055874, 2022–2025.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Zenodo Replication Package at https://doi.org/10.5281/zenodo.8043897.

ORCID

Sergio Di Martino https://orcid.org/0000-0002-1019-9004
Anna Rita Fasolino https://orcid.org/0000-0001-7116-019X
Luigi Libero Lucio Starace https://orcid.org/0000-0001-7945-9014
Porfirio Tramontana https://orcid.org/0000-0003-3264-185X

REFERENCES

1. Data.ai state of mobile 2022. https://www.data.ai/en/go/state-of-mobile-2022/, Last accessed: June 6, 2023; 2022.
2. Khalid H, Shihab E, Nagappan M, Hassan AE. What do mobile app users complain about? *IEEE Softw*. 2015;32(3):70-77.
3. Kaner C. A tutorial in exploratory testing. https://www.kaner.com/pdfs/QAIExploring.pdf; 2008.
4. Bach J. *The testing practitioner*: UTN Publishers; 2007.
5. Itkonen J, Mntyl MV, Lassenius C. The role of the tester's knowledge in exploratory software testing. *IEEE Trans Softw Eng*. 2013;39(5):707-724.
6. Robotium: User scenario testing for android – github repository. https://github.com/RobotiumTech/robotium, Seen on Oct. 22, 2022.; 2022.
7. Create ui tests with espresso test recorder. https://developer.android.com/studio/test/espresso-test-recorder, Seen on Oct. 22, 2022.; 2022.
8. Di Martino S, Fasolino AR, Starace LLL, Tramontana P. Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing. *Softw Test, Verif Reliab*. 2021;31(3):e1754. https://doi.org/10.1002/stvr.1754
9. Aniche M, Treude C, Zaidman A. How developers engineer test cases: An observational study. *IEEE Trans Softw Eng*. 2021;2021:1.
10. Zhang X, Feng Y, Liu D, Chen Z, Xu B. Research progress of crowdsourced software testing. *J Softw*. 2018;29(1):69-88.
11. Mao K, Capra L, Harman M, Jia Y. A survey of the use of crowdsourcing in software engineering. *J Syst Softw*. 2017;126:57-84. http://www.sciencedirect.com/science/article/pii/S0164121216301832
12. Hosseini M, Phalp K, Taylor J, Ali R. The four pillars of crowdsourcing: A reference model. In: 2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS), "International Conference on Research Challenges in Information Science (RCIS)" IEEE; 2014:1-12.
13. Wang Y, Papangelis K, Lykourentzou I, Khan V-J, Saker M, Yue Y, Grudin J. The dawn of crowdfarms. *Commun ACM*. 2022;65(8):64-70. https://doi.org/10.1145/3490698
14. Amazon mechanical turk. https://www.mturk.com/ Seen on Oct. 22, 2022.; 2022.
15. Release faster, with confidence - applause. https://www.applause.com/ Seen on Oct. 22, 2022.; 2022.
16. Utest - the professional network for testers. https://www.utest.com/ Seen on Oct. 22, 2022.; 2022.
17. Mooctest. http://www.mooctest.net/login2 Seen on Oct. 22, 2022.; 2022.
18. Testin. Testin. http://www.testin.net/ Seen on Oct. 22, 2022.; 2022.
19. Baidu crowd worker. https://test.baidu.com/ Seen on Oct. 22, 2022.; 2022.
20. Wang J, Wang S, Chen J, Menzies T, Cui Q, Xie M, Wang Q. Characterizing crowds to better optimize worker recommendation in crowdsourced testing. IEEE Transactions on Software Engineering; 2019.
21. van der Heiden G, Matson S. Market guide for application crowdtesting services *Edited by* Research G. https://www.gartner.com/en/documents/3890079; 2018.
22. Applause customer stories. https://www.applause.com/customers Seen on Oct. 22, 2022.; 2022.
23. Wang J, Yang Y, Krishna R, Menzies T, Wang Q. isense: Completion-aware crowdtesting management. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), International Conference on Software Engineering (ICSE). IEEE IEEE; 2019:912-923.
24. Wang J, Yang Y, Menzies T, Wang Q. isense2.0: Improving completion-aware crowdtesting management with duplicate tagger and sanity checker. *ACM Trans Softw Eng Methodol (TOSEM)*. 2020;29(4):1-27.
25. Wang J, Yang Y, Wang S, Hu Y, Wang D, Wang Q. Context-aware in-process crowdworker recommendation. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, International Conference on Software Engineering (ICSE). ACM/IEEE ACM/IEEE; 2020:1535-1546.
26. Mao K, Capra L, Harman M, Jia Y. A survey of the use of crowdsourcing in software engineering. *J Syst Softw*. 2017;126:57-84. https://www.sciencedirect.com/science/article/pii/S0164121216301832
27. Ge X, Yu S, Fang C, Zhu Q, Zhao Z. Leveraging android automated testing to assist crowdsourced testing. *IEEE Trans Softw Eng*. 2022;2022:1-18.

28. Di Martino S, Fasolino AR, Starace LLL, Tramontana P. Data and Code for the paper "An Empirical Study on Exploratory Crowdtesting of Android Applications". Zenodo, https://doi.org/10.5281/zenodo.8043897; 2023.
29. Abran A, Moore JW, Bourque P, Dupuis R, Tripp L. Software engineering body of knowledge. *IEEE Comput Soc, Angela Burgess*. 2004;2004:25.
30. Itkonen J, Mntyl MV. Are test cases needed? replicated comparison between exploratory and test-case-based software testing. *Empir Softw Eng*. 2014;19(2):303-342. cited By 25.
31. Itkonen J, Rautiainen K. Exploratory testing: a multiple case study. In: 2005 International Symposium on Empirical Software Engineering, 2005., International Symposium on Empirical Software Engineering IEEE; 2005:10-pp.
32. Whittaker JA. *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*: Pearson Education; 2009.
33. Wood M, Roper M, Brooks A, Miller J. Comparing and combining software defect detection techniques: A replicated empirical study. In: Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC'97/FSE-5. Springer-Verlag ACM; 1997; Berlin, Heidelberg:262-277. https://doi.org/10.1145/267895.267915
34. Basili VR, Selby RW. Comparing the effectiveness of software testing strategies. *IEEE Trans Softw Eng*. 1987;SE-13(12):1278-1296.
35. Kamsties E, Lott CM. An empirical evaluation of three defect-detection techniques. In: Proceedings of the 5th European Software Engineering Conference, European Software Engineering Conference. Springer-Verlag; 1995; Berlin, Heidelberg:362-383.
36. Andersson C, Thelin T, Runeson P, Dzamashvili N. An experimental evaluation of inspection and testing for detection of design faults. In: 2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. proceedings., International Symposium on Empirical Software Engineering; 2003:174-184.
37. Mao K, Harman M, Jia Y. Crowd intelligence enhances automated mobile testing. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), International Conference on Automated Software Engineering (ASE) IEEE/ACM; 2017:16-26.
38. Cui Q, Wang S, Wang J, Hu Y, Wang Q, Li M. Multi-objective crowd worker selection in crowdsourced testing. In: International Conference on Software Engineering and Knowledge Engineering, SEKE. Knowledge Systems Institute Graduate School Knowledge Systems Institute Graduate School; 2017:218-223.
39. Xie M, Wang Q, Yang G, Li M. Cocoon: Crowdsourced testing quality maximization under context coverage constraint. In: IEEE Computer Society IEEE; 2017:316-327.
40. Cui Q, Wang J, Yang G, Xie M, Wang Q, Li M. Who should be selected to perform a task in crowdsourced testing? In: International Computer Software and Applications Conference. IEEE IEEE; 2017:75-84.
41. Xie M, Wang Q, Cui Q, Yang G, Li M. Cqm: Coverage-constrained quality maximization in crowdsourcing test. Institute of Electrical and Electronics Engineers Inc. IEEE/ACM; 2017:192-194.
42. Wang J, Yang Y, Wang S, Hu J, Wang Q. Context-and fairness-aware in-process crowdworker recommendation. *ACM Trans Softw Eng Methodol*. 2022;31(3). cited By 0.
43. Wang J, Yang Y, Wang S, Chen C, Wang D, Wang Q. Context-aware personalized crowdtesting task recommendation. *IEEE Trans Softw Eng*. 2022;48(8):3131-3144. cited By 2.
44. Kamangar ZU, Kamangar UA, Ali Q, Farah I, Nizamani S, Ali TH. To enhance effectiveness of crowdsource software testing by applying personality types. Association for Computing Machinery ACM; 2019:15-19.
45. Nielsen J. Finding usability problems through heuristic evaluation. In: Proceedings of the Sigchi Conference on Human Factors in Computing Systems, CHI'92. Association for Computing Machinery ACM; 1992; New York, NY, USA:373-380. https://doi.org/10.1145/142750.142834
46. Sears A. Heuristic walkthroughs: Finding the problems without the noise. *Int J Human Comput Interaction*. 1997;9(3):213-234.
47. Wang J, Li M, Wang S, Menzies T, Wang Q. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Inf Softw Technol*. 2019;110:139-155. https://www.sciencedirect.com/science/article/pii/S0950584919300503
48. Robotium: User scenario testing for android – github repository. https://marketplace.eclipse.org/content/robotium-recorder, Seen on Oct. 22, 2022.; 2022.
49. Emma: a free java code coverage tool. http://emma.sourceforge.net/, Seen on Oct. 22, 2022.; 2022.
50. Emma reference manual. https://emma.sourceforge.net/userguide_single/userguide.html, Seen on Oct. 22, 2022.; 2022.
51. F-droid - free and open source android app repository. https://f-droid.org, Seen on Oct. 22, 2022.; 2022.
52. Amalfitano D, Amatucci N, Memon AM, Tramontana P, Fasolino AR. A general framework for comparing automatic testing techniques of android mobile apps. *J Syst Softw*. 2017;125:322-343. https://doi.org/10.1016/j.jss.2016.12.017
53. Choudhary SR, Gorla A, Orso A. Automated test input generation for android: Are we there yet?(e). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), International Conference on Automated Software Engineering (ASE) IEEE; 2015:429-440.
54. Mao K, Harman M, Jia Y. Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, International Symposium on Software Testing and Analysis ACM; 2016:94-105.
55. Munchlife: A munchkin level counter for android - github repository. https://github.com/pacebl/MunchLife, Seen on Oct. 22, 2022.; 2022.
56. Simply do – f-droid. https://f-droid.org/en/packages/kdk.android.simplydo/, Seen on Oct. 22, 2022.; 2022.
57. Tippy tipper (tip calculator) – github repository. https://github.com/mandlar/tippytipper, Seen on Oct. 22, 2022.; 2022.
58. Trolly – f-droid. https://f-droid.org/en/packages/caldwell.ben.trolly/, Seen on Oct. 22, 2022.; 2022.
59. Tramontana P, Amalfitano D, Amatucci N, Fasolino AR. Automated functional testing of mobile applications: a systematic mapping study. *Softw Qual J*. 2019;27(1):149-201. https://doi.org/10.1007/s11219-018-9418-6
60. Andrews JH, Groce A, Weston M, Xu R-G. Random test run length and effectiveness. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, International Conference on Automated Software Engineering IEEE/ACM; 2008:19-28.
61. Fraser G, Gargantini A. Experiments on the test case length in specification based test case generation. In: Proceedings of the 2009 ICSE Workshop on Automation of Software Test, AST 2009, ICSE Workshop on Automation of Software Test ACM/IEEE; 2009:18-26.
62. Arcuri A. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Trans Softw Eng*. 2012;38(3):497-519.
63. Xie Q, Memon AM. Studying the characteristics of a" good" gui test suite. In: 2006 17th International Symposium on Software Reliability Engineering, International Symposium on Software Reliability Engineering IEEE; 2006:159-168.
64. Wilcoxon F. Individual comparisons by ranking methods. *Breakthroughs Stat*. 1992;1992:196-202.

65. Cliff N. *Ordinal methods for behavioral data analysis*. New York: Psychology Press; 2014.
66. Kampenes VB, Dyb T, Hannay JE, Sjberg DIK. A systematic review of effect size in software engineering experiments. *Inf Softw Technol*. 2007;49(11): 1073-1086. https://www.sciencedirect.com/science/article/pii/S0950584907000195
67. Wohlin C, Runeson P, Host M, Ohlsson MC, Regnell B, Wesslen A. *Experimentation in software engineering*: Springer; 2012.
68. Zhong Y, Shi M, Xu Y, Fang C, Chen Z. Iterative android automated testing. *Front Comput Sci*. 2023;17(5):175212.
69. Carver J, Jaccheri L, Morasca S, Shull F. Issues in using students in empirical studies in software engineering education. In: Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE CAT. NO.03EX717), International Workshop on Enterprise Networking and Computing in Healthcare Industry IEEE; 2003:239-249.
70. Ciolkowski M, Muthig D, Rech J. Using academic courses for empirical validation of software development processes. In: Proceedings. 30th EUROMICRO Conference, 2004. Euromicro; 2004:354-361.
71. Host M, Regnell B, Wohlin C. Using students as subjects a comparative study of students and professionals in lead-time impact assessment. *Empir Softw Eng*. 2000;5(3):201-214.
72. Chen Z, Luo B. Quasi-crowdsourcing testing for educational projects. In: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014. Association for Computing MachineryIEEE/ACM; 2014; New York, NY, USA:272-275. https://doi.org/10.1145/2591062.2591153
73. Salman I, Misirli AT, Juristo N. Are students representatives of professionals in software engineering experiments? In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE International Conference on Software Engineering, vol. 1 IEEE; 2015:666-676.

How to cite this article: Di Martino S, Fasolino AR, Starace LL, Tramontana P. GUI testing of Android applications: Investigating the impact of the number of testers on different exploratory testing strategies. *J Softw Evol Proc*. 2024;36(7):e2640. doi:10.1002/smr.2640