Automated Testing of Android Apps: A Systematic Literature Review

Pingfan Kong, Li Li[ξ], Jun Gao, Kui Liu, Tegawendé F. Bissyandé, Jacques Klein

Abstract—Automated testing of Android apps is essential for app users, app developers and market maintainer communities alike. Given the widespread adoption of Android and the specificities of its development model, the literature has proposed various testing approaches for ensuring that not only functional requirements but also non-functional requirements are satisfied. In this paper, we aim at providing a clear overview of the state-of-the-art works around the topic of Android app testing, in an attempt to highlight the main trends, pinpoint the main methodologies applied and enumerate the challenges faced by the Android testing approaches as well as the directions where the community effort is still needed. To this end, we conduct a Systematic Literature Review (SLR) during which we eventually identified 103 relevant research papers published in leading conferences and journals until 2016. Our thorough examination of the relevant literature has led to several findings and highlighted the challenges that Android testing researchers should strive to address in the future. After that, we further propose a few concrete research directions where testing approaches are needed to solve recurrent issues in app updates, continuous increases of app sizes, as well as the Android ecosystem fragmentation.

1 Introduction

Android smart devices have become pervasive after gaining tremendous popularity in recent years. As of July 2017, Google Play, the official app store, is distributing over 3 million Android applications (i.e., apps), covering over 30 categories ranging from entertainment and personalisation apps to education and financial apps. Such popularity among developer communities can be attributed to the accessible development environment based on familiar Java programming language as well as the availability of libraries implementing diverse functionalities [1]. The app distribution ecosystem around the official store and other alternative stores such as Anzhi and AppChina is further attractive for users to find apps and organisations to market their apps [2].

Unfortunately, the distribution ecosystem of Android is porous to poorly-tested apps [3]–[5]. Yet, as reported by Kochhar [3], error-prone apps can significantly impact user experience and lead to a downgrade of their ratings, eventually harming the reputation of app developers and their organizations [5]. It is thus becoming more and more important to ensure that Android apps are sufficiently tested before they are released on the market. However, instead of manual testing, which is often laborious, time-consuming and error-prone, the ever-growing complexity and the enormous number of Android apps call for scalable, robust and trustworthy automated testing solutions.

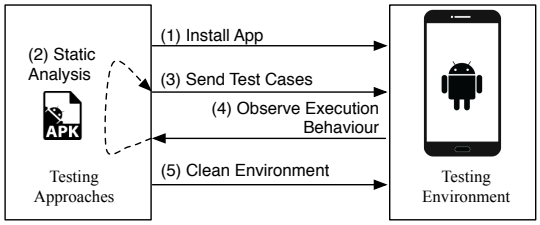


Fig. 1: Process of testing Android apps.

Android app testing aims at testing the functionality, usability and compatibility of apps running on Android devices [6], [7]. Fig. 1 illustrates a typical working process. At Step (1), target app is installed on an Android device. Then in Step (2), the app is analysed to generate test cases. We remind the readers that this step (in dashed line) is optional and some testing techniques such as automated random testing do not need to obtain pre-knowledge for generating test cases. Subsequently, in Step (3), these test cases are sent to the Android device to exercise the app. In Step (4), execution behaviour is observed and collected from all sorts of perspectives. Finally, in Step (5), the app is uninstalled and relevant data is wiped. We would like to remind the readers that installation of the target app is sometimes not a necessity, e.g., frameworks like Robolectric allow tests directly run in JVM. In fact, Fig. 1 can be borrowed to describe the workflow of testing almost any software besides Android apps. Android app testing, on the contrary, falls in a unique context and often fails to use general testing techniques [8]–[13]. There are several differences with traditional (e.g., Java) application testing that motivate research on Android app testing. We enumerate and consider for our review a few common challenges:

First, although apps are developed in Java, traditional Java-based testing tools are not immediately usable on An-

- [ξ] *The corresponding author.*
- *P. Kong, J. Gao, K. Liu, T. Bissyandé, and J. Klein are with the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg.*
- *L. Li is with the Faculty of Information Technology, Monash University, Australia.*
  E-mail: li.li@monash.edu

Manuscript received XXX; revised XXX.This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects CHARACTERIZE C17/IS/11693861 and Recommend C15/IS/10449467.

droid apps since most control-flow interactions in Android are governed by specific event-based mechanisms such as the Inter-Component Communication (ICC [14]). To address this first challenge, several new testing tools have been specifically designed for taking Android specificities into account. For example, RERAN [15] was proposed for testing Android apps through a timing- and touch-sensitive record-and-replay mechanism, in an attempt to capture, represent and replay complicated non-discrete gestures such as *circular bird swipe with increasing slingshot tension in Angry Birds*.

Second, Android fragmentation, in terms of the diversity of available OS versions and target devices (e.g., screen size varieties), is becoming acuter as now testing strategies have to take into account different execution contexts [16], [17].

Third, the Android ecosystem attracts a massive number of apps requiring scalable approaches to testing. Furthermore, these apps do not generally come with open source code, which may constrain the testing scenarios.

Finally, it is challenging to generate a perfect coverage of test cases, in order to find faults in Android apps. Traditional test case generation approaches based on *symbolic execution* and tools such as *Symbolic Pathfinder (SPF)* are challenged by the fact that Android apps are available in Dalvik bytecode that differs from Java bytecode. In other words, traditional Java-based symbolic execution approaches cannot be directly applied to tackle Android apps. Furthermore, the event-driven feature, as well as framework libraries, pose further obstacles for systematic generation of test cases [18].

Given the variety of challenges in testing Android apps, it is important for this field, which has already produced a significant amount of approaches, to reflect on what has already been solved, and on what remains to tackle. To the best of our knowledge, there is no related literature review or survey summarizing the topic of Android testing. Thus, we attempt to meet this need through a comprehensive study. Concretely, we undertake a systematic literature review (SLR), carefully following the guidelines proposed by Kitchenham et al. [19] and the lessons learned from applying SLR within the software engineering domain by Brereton et al. [20]. To achieve our goal, we have searched and identified a set of relevant publications from four well-known repositories including the ACM Digital Library and from major testing-related venues such as ISSTA, ICSE. Then, we have performed a detailed overview on the current state of research in testing Android apps, focusing on the types and phases of the testing approaches applied as well as on a trend analysis in research directions. Eventually, we summarize the limitations of the state-of-the-art apps and highlight potential new research directions.

The main contributions of this paper are:

- We build a comprehensive repository tracking the research community effort to address the challenges in testing Android apps. In order to enable an easy navigation of the state-of-the-art, thus enabling and encouraging researchers to push the current frontiers in Android app testing, we make all collected and built information publicly available at

  http://lilicoding.github.io/TA2Repo/

- We analyse in detail the key aspects in testing Android apps and provide a taxonomy for clearly summarising and categorising all related research works.
- Finally, we investigate the current state of the art, enumerate the salient limitations and pinpoint a few directions for furthering the research in Android testing.

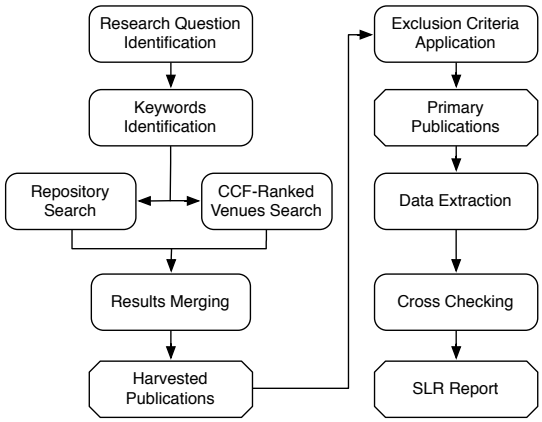


Fig. 2: Process of the SLR.

The rest of the paper is organized as follows: Section 2 depicts the methodology of this systematic literature review, including a general overview and detailed reviewing processes of our approach. In Section 3, we present the results of our selected primary publications, along with a preliminary trend and statistic analysis on those collected publications. Later, we introduce our data extraction strategy and their corresponding findings in the following two sections: Section 4 and 5. After that, we discuss the trends we observed and challenges the community should attempt to address in Section 6 and enumerate the threats to validity of this SLR in Section 7. A comparison of this work with literature studies is given in Section 8 and finally we conclude this SLR in Section 9.

2 Methodology of This SLR

We now introduce the methodology applied in this SLR. We remind the readers that an SLR follows a well-defined strategy to systematically identify, examine, synthesize, evaluate and compare all available literature works in a specific topic, resulting in a reliable and replicable report [19], [21], [22]. Fig. 2 illustrates the process of our SLR. At the beginning, we define relevant research questions (cf. Section 2.1) to frame our investigations. The following steps are unfolded to search and consolidate the relevant literature, before extracting data for answering the research questions, and finalizing the report.

Concretely, to harvest all relevant publications, we identify a set of search keywords and apply them in two separate processes: 1) online repository search and 2) major[1] venues search. All results are eventually merged for further reviewing (cf. Section 2.2). Next, we apply some exclusion criteria on the merged list of publications, to exclude irrelevant papers (e.g., papers not written in English) or less relevant papers (e.g., short papers), in order to focus on a small, but

1. We rely on the China Computer Federation (CCF) ranking of computer science venues.

highly relevant, set of primary publications (cf. Section 2.3). Finally, we have developed various metrics and reviewed the selected primary publications against these metrics through full paper examination. After the examination, we cross-check the extracted results to ensure their correctness and eventually we report on the findings to the research community (cf. Section 2.4).

2.1 Initial research questions

Given the common challenges enumerated in the Introduction section, which have motivated several research lines in Android apps, we investigate several research questions to highlight how and which challenges have been focused on in the literature. In particular, with regards to the fact that Android has programming specificities (e.g., event-based mechanisms, GUI), we categorize test concerns targeted by the research community. With regards to the challenge of ensuring scalability, we study the tests levels which are addressed in research works. With regards to the challenge of generating test cases, we investigate in details the fundamental testing techniques leveraged. Finally, with regards to the fragmentation of the Android ecosystem, we explore the extent of validation schemes for research approaches. Overall, we note that testing Android apps is a broad activity that can target a variety of functional and non-functional requirements and verification issues, leverage different techniques and focus on different granularity levels and phases. Our investigation thus starts with the following related research questions:

- **RQ1: What are the test concerns?** With this research question, we survey the various objectives sought by Android app testing researchers. In general, we investigate the testing objectives at a high level to determine what requirements (e.g., security, performance, defects, energy) the literature addresses. We look more in-depth into the specificities of Android programming, to enumerate the priorities that are tackled by the community, including which concerns (e.g., GUI and ICC mechanism) are factored in the design of testing strategies.
- **RQ2: Which test levels are addressed?** With the second research question, we investigate the levels (i.e., when the tests are relevant in the app development process) that research works target. The community could indeed benefit from knowing to what extent regression testing is (or is not) developed for apps which are now commonly known to evolve rapidly.
- **RQ3: How are the testing approaches built?** In the third research question, we process detailed information on the design and implementation of test approaches. In particular, we investigate the fundamental techniques (e.g., concolic testing or mutation testing) leveraged, as well as the amount of input information (i.e., to what extent the tester should know about the app prior to testing) that approaches require to perform.
- **RQ4: To what extent are the testing approaches validated?** Finally, the fourth research question investigates the metrics, datasets and procedures in the literature for measuring the effectiveness of state-of-the-art approaches. Answers to this question may shed light on the gaps in the research agenda of Android testing.

2.2 Search Strategy

We now detail the search strategy that we applied to harvest literature works related to Android app testing.

Identification of search keywords. Our review focuses on two key aspects: Testing and Android. Since a diversity of terms may be used by authors to refer, broadly or precisely, to any of these aspects, we rely on the extended set of keywords identified in Table 1. Our final search string is then constructed as a conjunction of these two categories of keywords ($search\_string = cat1 \ \& \ cat2$), where each category is represented as a disjunction of its keywords ($cat = kw1 \mid kw2 \mid kw3$).

TABLE 1: Search Keywords

Category	Keywords
Android	android, mobile, portable device, smartphone, smart phone, smart device
Test	test, testing, measure, measurement, measuring, check, checking, detect, detecting, detection

Online repository search. We use the search string on online literature databases to find and collect relevant papers. We have considered four widely used repository for our work: ACM Digital Library[2], IEEE Xplore Digital Library[3], SpringerLink[4], and ScienceDirect[5]. The "advanced" search functionality of the four selected online repositories are known to be inaccurate, which usually result in a huge set of irrelevant publications, noising the final paper set [22]. Indeed, those irrelevant publications do not really match our keywords criteria. For example, they may not contain any of the keywords shown in the *Test* category. Thus, we develop scripts (combined with Python and Shell) to perform off-line matching verification on the papers yielded by those search engines, where the scripts follow exactly the same criteria that we have used for online repository search. For example, regarding the keywords enumerated in the *Test* category, if none of them is presented in a publication, the scripts will mark that publication as irrelevant and subsequently exclude it from the candidate list.

Major venues search. Since we only consider a few repositories for search, the coverage can be limited given that a few conferences such as NDSS[6] and SEKE[7] do not host their proceedings in the aforementioned repositories. Thus, to mitigate the threat to validity of not including all relevant papers, we further explicitly search in proceedings of all major venues in computer science. We have chosen the comprehensive CCF-ranking of venues[8] and leveraged the DBLP[9] repository to collect the Document Object Identifiers (DOI) of the publications in order to crawl abstracts and all publication metadata. Since this search process considers major journal and conference venues, the resulting set of

2. http://dl.acm.org/
3. http://ieeexplore.ieee.org/Xlpore/home.jsp
4. http://link.springer.com
5. http://www.sciencedirect.com
6. The Network and Distributed System Security Symposium
7. International Conference on Software Engineering & Knowledge Engineering
8. http://www.ccf.org.cn/sites/ccf/paiming.jsp, we only take into account *software engineering* and *security* categories, as from what have observed, the majority of papers related to *testing Android apps.*
9. http://dblp.uni-trier.de

literature papers should be a representative collection of the state-of-the-art.

2.3 Exclusion Criteria

After execution of our search based on the provided keywords, a preliminary manual scanning showed that the results are rather coarse-grained since it included a number of irrelevant or less relevant publications which, nonetheless, matched[10] the keywords. It is thus necessary to perform a fine-grained inclusion/exclusion in order to focus on a consistent and reliable set of primary publications and reduce the eventual effort in further in-depth examination. For this SLR, we have applied the following exclusion criteria:

1) Papers that are not written in English are filtered out since English is the common language spoken in the worldwide scientific peer-reviewing community.
2) Short papers are excluded, mainly because such papers are often work-in-progress or idea papers: on the one hand, short papers are generally not mature, and, on the other hand, many of them will eventually appear later in a full paper format. In the latter case, mature works are likely to already be included in our final set. In this work, we take a given publication as a short paper when it has fewer than 4 pages (included) in IEEE/ACM-like double-column format [11] or fewer than 8 pages (included) in LNCS-like single column format as short papers are likely to be 4 pages in double column format and 8 pages in single column format.
3) Papers that are irrelevant to testing Android apps are excluded. Our search keywords indeed included broad terms such as *mobile* and *smartphone* as we aimed at finding all papers related to Android even when the term "Android" was not specifically included in the title and abstract. By doing so, we have excluded papers that only deal with mobile apps for other platforms such as iOS and Windows.
4) Duplicated papers are removed. It is quite common for authors to publish an extended version of their conference paper to a journal venue. However, these papers share most of the ideas and approach steps. To consider both of them would result in a biased weighting of the metrics in the review. To mitigate this, we identify duplicate papers by first comparing paper titles, abstracts and authors and then further manually check when a given pair of records share a major part of their contents. We filter out the least recent publication when duplication is confirmed.
5) Papers that conduct comparative evaluations, including surveys on different approaches of testing Android apps, are excluded. Such papers indeed do not introduce new technical contributions for testing Android apps.
6) Papers in which the testing approach targets the operating system, networks, or hardware, rather than mobile apps are excluded.
7) Papers that assess[12] existing testing methods are also filtered out. The publications that they discuss are supposed to be already included in our search results.
8) Papers demonstrating how to set up environments and platforms to retrieve runtime data from Android apps are excluded. These papers are also important for Android Apps testing, but they are not focusing on new testing methodology.
9) Finally, some of our keywords (e.g., "detection" of issues, "testing" of apps) have led to the retrieval of irrelevant literature works that must be excluded. We have mainly identified two types of such papers: the first includes papers that perform *detection* of malicious apps using machine learning (and not testing); the second includes papers that describe the building of complex platforms, adopting existing mature *testing* methodologies.

We refer to all collected papers that remain after the application of exclusion criteria as **primary publications**. These publications are the basis for extracting review data.

2.4 Review Protocol

Concretely, the review is conducted in two phases: 1) First, we perform an abstract review and quick full paper scan to filter out irrelevant papers based on the exclusion criteria defined above. At the end of this phase, the set of primary publications is known. 2) Subsequently, we perform a full review of each primary publication and extract relevant information that is necessary for answering all of our research questions.

In practice, we have split our primary publications to all the co-authors to conduct the data extraction step. We have further cross-checked all the extracted results: when some results are in disagreement, informal discussions are conducted until a consensus is reached.

3 Primary Publications Selection

TABLE 2: Summary of the selection of primary publications.

Step	Count
Repository and Major Venues Search	9259
After reviewing titles/abstracts (scripts)	472
After reviewing titles/abstracts	255
After skimming/scanning full paper	171
After final discussion	103

Table 2 summarizes statistics of collected papers during the search phase. Overall, our repository search and major venue search have yielded in total 9,259 papers.

Following the exclusion criteria in Section 2, the papers satisfying the matching requirements immediately drop from 9259 to 472. We then manually go through the title and abstract of each paper to further dismiss those that match the exclusion criteria. After this step, the set of papers is reduced to 255 publications. Subsequently, we go through the full content of papers in the set, leading to the exclusion of 84 more papers. Finally, after discussion among the authors for the rest of the set, we reach a consensus on considering 103 publications as relevant primary publications.

10. The keywords were found for example to be mentioned in the related sections of the identified papers.

11. Note that we have actually kept a short paper entitled "GuiDiff: a regression testing tool for graphical user interface" because it is very relevant to our study and it does not have an extended version released in the following years.

12. For example, [23] and [24] propose tools and algorithms for measuring the code coverage of testing methods.

Fig. 3: Word Cloud based on the Venue Names of Selected Primary Publications.

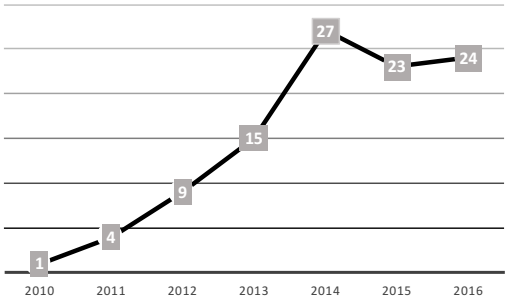


Fig. 4: The number of publications in each year.

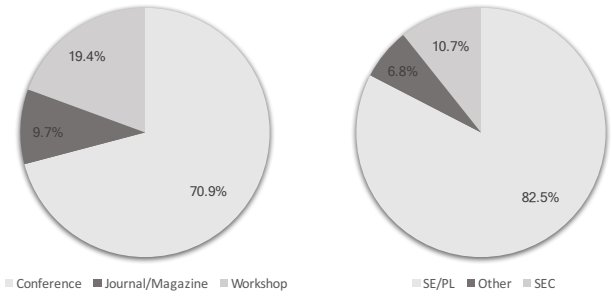
Table A1 (in appendix) enumerates the details of those 103 publications.

It is noteworthy that around 4% of the final primary publications are exclusively found by major venues search, meaning that they cannot be found based on well-known online repositories such as IEEE and ACM. This result, along with our previous experiences [22], suggests that repository search is necessary but not sufficient for harvesting review publications. Other steps (e.g., top venues search based on Google Scholar impact factor [22] or CCF ranking) should be taken in complement to ensure reliable coverage of state-of-the-art papers.

Fig. 3 presents a word cloud based on the venue names of selected primary publications. The more papers selected from a venue, the bigger its name showing in the word cloud. Not surprisingly, the recurrently targeted venues are mainly testing-related conferences such as ISSTA, ICST, ISSRE, etc.

Fig. 4 illustrates the trend of the number of publications in each year we have considered. From this figure, we can observe that the number of papers tackling the problem of testing Android apps has increased gradually to reach a peak in 2014. Afterwards, the pace of developing new testing techniques has stabilized.

We further look into the selected primary publications through their published venue types and domains. Fig. 5a and Fig. 5b illustrate the statistic results, respectively. Over 90% of examined papers are published in conferences and workshops (which are usually co-located with top conferences) while only 10% papers are published in journals. These findings are in line with the current situation where intense competition in Android research forces researchers



(a) Venue Types. (b) Venue Domains.

Fig. 5: Distribution of examined publications through published venue types and domains.

to make available their works as fast as possible. We further find that over 80% of examined papers are published in software engineering and programming language venues, showing that testing Android apps is mainly a concern in the software engineering community. Nevertheless, as shown by several papers published in proceedings of security venues, testing is also a valuable approach to address security issues in Android apps.

4 Taxonomy of Android Testing Research

To extract relevant information from the literature, our SLR must focus on specific characteristics eventually described in each publication. To facilitate this process in a field that explores a large variety of approaches, we propose to build a taxonomy of Android testing. Such a taxonomy eventually helps to gain insights into the state-of-the-art by answering the research questions proposed in Section 2.1.

By searching for answers to the aforementioned research questions in each publication, we are able to make a systematic assessment of the literature with a schema for classifying and comparing different approaches. Fig. 6 presents a high-level view of the taxonomy diagram spreading in four dimensions (i.e., **Test Objectives**, **Test Targets**, **Test Levels** and **Test Techniques**) associated with the first three research questions[13].

Test Objectives. This dimension summarizes the targeted objectives of our examined testing-related publications. We have enumerated overall 6 recurring testing objectives such as Bug/Defect detection.

Test Targets. This dimension summarizes the representative targets where testing approaches focus on. In particular, for testing Android apps, the GUI/Event and ICC/IAC are recurrently targeted. For simplicity, we regroup all the other targets such as normal code analysis into *General*.

Test Levels. This dimension checks the different levels (also known as phases) at which the test activities are performed. Indeed, there is a common knowledge that software testing is very important and has to be applied to many levels such as *unit testing*, *integration testing*, etc. Android apps, as a specific type of software, also need to go through

13. **Test Objectives** and **Test Targets** for RQ1 (test concerns), **Test Levels** for RQ2 (test levels) and **Test Techniques** for RQ3 (test approaches). RQ4 explores the validity of testing approaches that is not summarised in the taxonomy.

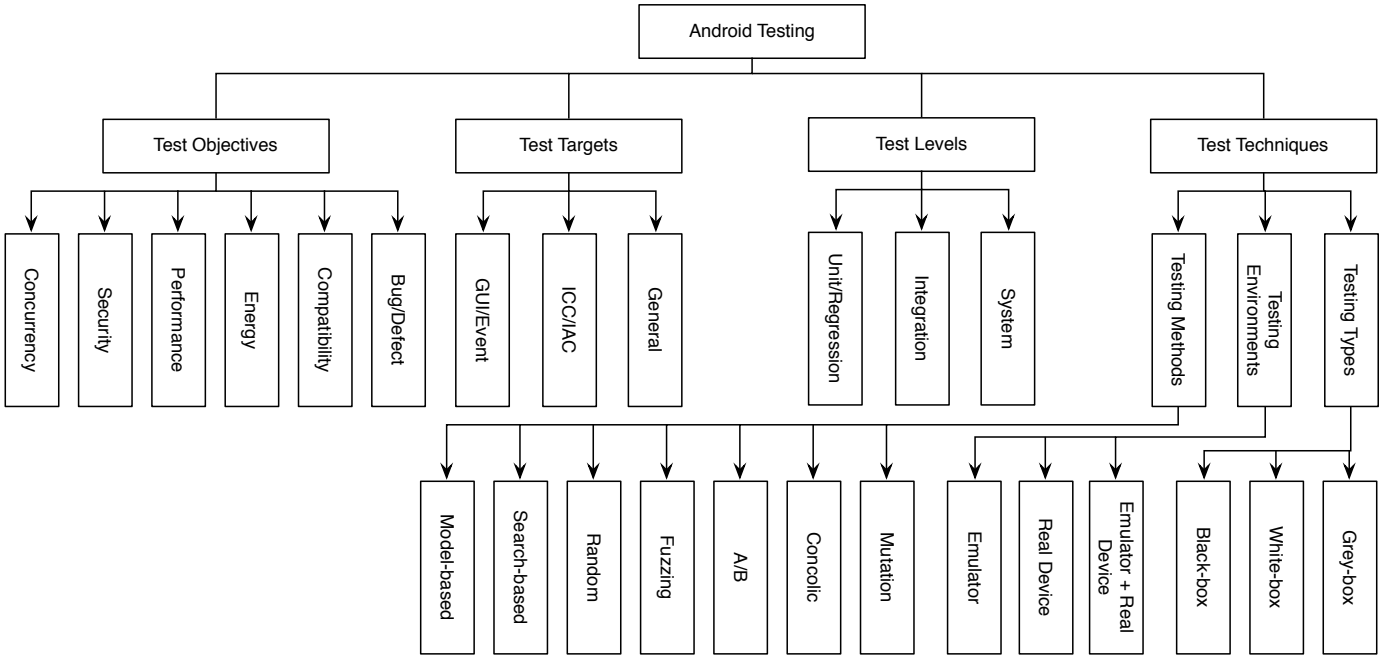


Fig. 6: Taxonomy of Android App Testing.

a thorough testing progress before being released to public markets. In this dimension, we sum up the targeted testing phases/levels of examined approaches, to understand what has been focused so far by the state-of-the-art.

Test Techniques. Finally, the fourth dimension focuses on the fundamental methodologies (e.g., Fuzzy or Mutation) that are followed to perform the tests, as well as the testing environments (e.g., on emulated hardware) and testing types (e.g., black-box testing).

5 Literature Review

We now report on the findings of this SLR in light of the research questions that we have raised in Section 2.2.1.

5.1 What concerns do the approaches focus on?

Our review investigates both the objectives that testing approaches seek to achieve and the app elements that are targeted by the test cases. *Test objectives* focus on problems that can be located anywhere in the code, while *test targets* focus on specific app elements that normally involve only certain types of code (e.g., functionality).

5.1.1 Test objectives

Android testing research has tackled various objectives, including the assessment of apps against non-functional properties such as app efficiency in terms of *energy* consumption, and functional requirements such as the presence of bugs. We discuss in this section some recurrent test objectives from the literature.

Concurrency. Android apps expose a concurrency model that combines multi-threading and asynchronous event-based dispatch, which may lead to subtle concurrency errors because of unforeseen thread interleaving coupled with non-deterministic reordering of asynchronous tasks. These error-prone features are however useful and increasingly becoming common in the development of efficient and feature-rich apps. To mitigate concurrency issues, several works have been proposed, notably for detecting races such as data races, event-based races, etc. in Android apps. As an example, Maiya et al. [62] have built DroidRacer, which identifies data races (i.e., the *read* and *write* operations happen in parallel) by computing the happens-before relation on execution traces that are generated systematically through running test scenarios against Android apps. Bielik et al. [47] later have proposed a novel algorithm for scaling the inference of happens-before relations. Hu et al. [9] present a work for verifying and reproducing event-based races, where they have found that both imprecise Android component modelling and implicit happens-before relation could result in false positive for detecting potential races.

Security. As shown by Li et al. [22], the Android research community is extensively working on providing tools and approaches for solving various security problems for Android apps. Some of these works involve app testing, e.g., to observe defective behaviour [57] and malicious behaviour [79], track data leaks [75]. For example, Yan et al. [78] have built a novel and comprehensive approach for the detection of resource leaks using test criteria based on neutral cycles: sequences of GUI events should have a "neutral" effect and should not increase the usage of resources. Hay et al. [45] dynamically detect inter-application communication vulnerabilities in Android apps.

Performance. Android apps are sensitive to performance issues. When a program thread becomes expensive, the system may stop app execution after warning on the user interface that the "Application [is] Not Responding". The literature includes several contributions on highlighting issues related to the performance of Android apps such as poor responsiveness [29] and exception handling [55]. Yang et al. [74], for example, have proposed a systematic testing approach to uncover and quantify common causes of poor

TABLE 3: Test objectives in the literature.

Tool	Concurrency	Security	Performance	Energy	Compatibility	Bug/Defect	Tool	Concurrency	Security	Performance	Energy	Compatibility	Bug/Defect
Dagger [25]		✓					Malisa et al. [26]		✓				
CRASHSCOPE [27]						✓	MAMBA [28]		✓				
Pretect [29]			✓				SSDA [30]						✓
TrimDroid [8]						✓	ERVA [9]	✓					
SAPIENZ [11]						✓	RacerDroid [31]	✓					✓
DiagDroid [32]			✓				MOTIF [33]						✓
DRUN [34]	✓						GAT [35]						✓
Zhang et al. [36]						✓	Jabbarvand et al. [37]				✓		✓
Qian et al. [38]			✓			✓	Ermuth et al. [39]		✓				
Zhang et al. [40]			✓				Zhang et al. [41]					✓	
dLens [42]				✓			Packeviius et al. [43]						✓
Knorr et al. [44]		✓					IntentDroid [45]		✓				
Farto et al. [46]			✓				Bielik et al. [47]	✓					
MobiGUITAR [48]						✓	Aktouf et al. [49]						✓
AppAudit [50]						✓	Hassanshahi et al. [51]		✓				
iMPAcT [52]						✓	Deng et al. [53]						✓
Espada et al. [54]				✓			Zhang et al. [55]			✓			
QUANTUM [56]						✓	CRAXDroid [57]		✓	✓			✓
IntentFuzzer [58]		✓				✓	Vikomir et al. [59]					✓	
Shahriar et al. [60]		✓	✓			✓	APSET [61]		✓				
DROIDRACER [62]	✓						AppACTS [63]					✓	
CAFA [64]	✓						Guo et al. [65]		✓				
Griebe et al. [66]						✓	PBGT [67]					✓	
Banerjee et al. [68]				✓			A5 [69]		✓				
Suarez et al. [70]		✓					Linares et al. [71]				✓		
Sasnauskas et al. [72]						✓	AMDetector [73]		✓				
RERAN [15]						✓	Yang et al. [74]			✓			✓
DroidTest [75]		✓					Appstrument [76]			✓			
Avancini et al. [77]		✓					LEAKDROID [78]			✓			
Mahmood et al. [79]		✓	✓				Franke et al. [80]	✓					
Dhanapal et al. [81]			✓				SmartDroid [82]						✓
JarJarBinks [83]						✓	Hu et al. [84]						✓
Count								7	18	13	5	4	27

responsiveness of Android apps. Concretely, they explicitly extend the delay for typical problematic operations, using the test amplification approach, to demonstrate the effects of expensive actions that can be observed by users.

Energy. One of the biggest differences between traditional PC and portable devices is the fact that portable devices may run on battery power which can get depleted during app usage. A number of research works have investigated energy consumption hotspots arising from software design defects, unwanted *service* execution (e.g., advertisement), or have leveraged energy fingerprints to detect mobile malware. As an example, Wan et al. [42] present a technique for detecting display energy hotspots to guide the developers to improve the energy efficiency of their apps. Since each activity performed on a battery powered device drains a certain amount of energy from it, if the normal energy consumption is known for a device, the additionally used energy should be flagged as abnormal.

Compatibility. Android apps are often suffering from compatibility issues, where a given app can run successfully on a device, characterized by a range of OS versions while failing on others [85]. This is mainly due to the fragmentation in the Android ecosystem brought by its open source nature. Every vendor, theoretically, can have its own customized system (e.g., for supporting specific low-level hardware) and the screen size of its released devices can vary as well. To address compatibility problems, there is a need to devise scalable and efficient approaches for performing compatibility testing before releasing an app into markets. Indeed, as pointed out by Vilkomir et al. [59], it is expensive and time-consuming to consider testing all device variations. The authors thus proposed to address the issue with a combinatorial approach, which attempts to select an optimal set of mobile devices for practical testing. Zhang et al. [41] leverage a statistical approach to optimize the compatibility testing strategy where the test sequence is generated by K-means statistic algorithm.

Bug/Defect[14]**.** Like most software, Android apps are often buggy, usually leading to runtime crashes. Due to the high competition of apps in the Android ecosystem, defect identification is critical since they can be detrimental to user rating and adoption [86]. Indeed, researchers in this field leverage various testing techniques such as fuzzing testing,

14. Terminologically, the aforementioned objectives could also be categorised as bug/defect problems (e.g., concurrency issues). To make the summarisation more meaningful in this work, we only flag publications as bug/defect as long as their main focuses are bug/defect problems, e.g., when they address the gap between app's misbehaviour and developer's original design.

mutation testing, and search-based testing to dynamically explore Android apps to pinpoint defective behaviour [57], GUI bugs [84], Intent defects [72], crashing faults [11], etc.

Table 3 characterizes the publications selected for our SLR in terms of the objectives discussed above. Through our in-depth examination, the most considered testing objective is bug/defect, accounting for 23.3% of the selected publications.

5.1.2 Test targets

Test approaches in software development generally target core functionality code. Since Android apps are written in Java, the literature on Android app testing focused on Android specificities, mainly on how to address the GUI testing with a complex event mechanism as well as inter-component and inter-application communications.

GUI/Event. Android implements an event-driven graphical user interface system, making Android apps testing challenging, since they intensively interact with user inputs, introducing uncertainty and non-determinism. It is generally complicated to model the UI/system events because it not only needs the knowledge of the set of GUI widgets and their supporting actions (e.g., click for buttons) but also requires the knowledge of system events (e.g., receiving a phone call) which however are usually unknown in advance. Consequently, it is generally difficult to assemble a valid set of input event sequences for a given Android app with respect to *coverage*, *precision*, and *compactness* test criteria [87]. The Android testing community has proposed many approaches to address this challenge. For example, Android-GUITAR, an extension of the GUITAR tool [88] was proposed to model the structure and execution behaviour of Android GUI through a formalism called GUI forests and event-flow graphs. Denodroid [89] applies a dynamic approach to generate inputs by instrumenting the Android framework to record the reaction of events.

ICC/IAC. The Inter-Component Communication (ICC) and Inter-Application communication (IAC[15]) enable a loose coupling among components [90], [91], thus reducing the complexity to develop Android apps with a generic means to reuse existing functionality (e.g., obtain the contact list). Unfortunately, ICC/IAC also come with a number of security issues, among which the potential for implementing component hijacking, broadcast injection, etc. [92]. Researchers have then investigated various testing approaches to highlight such issues in Android apps. IntentDroid [45], for instance, performs comprehensive IAC security testing for inferring Android IAC integrity vulnerabilities. It utilizes lightweight platform-level instrumentation, which is implemented through debug breakpoints, to recover IAC-relevant app-level behaviour. IntentFuzzer [58], on the other hand, leverages fuzz testing techniques to detect capability leaks (e.g., permission escalation attacks) in Android apps.

General For all other publications which did not address the above two popular targets, the category *General* applies. Publications with targets like normal code analysis are grouped into this category.

Table 4 characterizes the test targets discussed above. The most frequently addressed testing target is GUI/Event, accounting for 45.6% of the selected publications. Meanwhile, there are only 12 publications targeted ICC/IAC. 44 publications are regrouped under the *General* category.

15. IAC is actually ICC where the communicating components are from different apps.

Insights from RQ1 - on Targets and Objectives

– "Bug/defect" has been the most trending concern among Android research community. "Compatibility" testing which is necessary for detecting issues that plague the Android fragmented ecosystem remains understudied. Similarly, we note that because mobile devices are quickly getting powerful, developers build increasingly complex apps with services exploring hardware multi-core capabilities. Therefore, the community should invest more efforts in approaches for concurrency testing.
– Our review has also confirmed that GUI is of paramount importance in modern software development for guaranteeing a good user experience. In Android apps, the GUI actions and reactions are intertwined with the app logic, increasing the challenges of analysing app codes for defects. For example, modelling GUI behaviour while taking into account potential runtime interruption by system events (e.g., incoming phone call) is necessary, yet not trivial. These challenges have created opportunities in Android research: as our literature review shows, most test approaches target GUI or the Event mechanism. The community now needs to focus on transforming the approaches into scalable tools that will perform deeper security analyses and accurate defect identification in order to improve the overall quality of apps distributed in markets.

5.2 Which Test Levels are Addressed?

Development of Android apps involves classical steps of traditional software development. Therefore, there are opportunities in various phases to perform tests with specific emphasis and purpose. The Software testing community commonly acknowledges four levels of software testing [127], [128]. Our literature review has identified that Android researchers have proposed approaches which considered *Unit/Regression testing*, *Integration testing*, and *System testing*. *Acceptance testing*, which involves end-users evaluating whether the app complies with their needs and requirements, still faces a lack of research effort in the literature.

Unit Testing is usually applied at the beginning of the development of Android apps, which are usually written by developers and can be taken as a type of white-box testing. Unit testing intends to ensure that every functionality, which could be represented as a function or a component, works properly (i.e., in accordance with the test cases). The main goal of unit testing is to verify that the implementation works as intended. Regression testing consists in re-executing previously executed test cases to ensure that subsequent updates of the app code have not impacted the original program behaviour, allowing issues (if presented) to be resolved as quickly as possible. Usually, regression testing is based on unit testing. It re-executes all the unit test cases every time when a piece of code is changed. As an example, Hu et al. [84] have applied unit testing to

TABLE 4: Test targets in the literature.

Tool	GUI/Event	ICC/IAC	General	Tool	GUI/Event	ICC/IAC	General
Zeng et al. [12]	✓			Dagger [25]			✓
Malisa et al. [26]	✓			CRASHSCOPE [27]	✓		
MAMBA [28]			✓	Pretect [29]	✓		
DroidMate [93]	✓			SSDA [30]			✓
TrimDroid [8]	✓			ERVA [9]			✓
Clapp et al. [10]	✓			SAPIENZ [11]			✓
RacerDroid [31]			✓	Baek et al. [94]	✓		
DiagDroid [32]	✓			MobiPlay [95]	✓		
MOTIF [33]			✓	DRUN [34]			✓
DroidDEV [96]	✓			GAT [35]	✓		
Zhang et al. [36]	✓			Jabbarvand et al. [37]			✓
Qian et al. [38]			✓	Ermuth et al. [39]			✓
Cadage [97]	✓			Zhang et al. [40]			✓
Zhang et al. [41]			✓	dLens [42]			✓
Sonny et al. [98]			✓	Packeviius et al. [43]	✓		
SIG-Droid [99]			✓	Knorr et al. [44]			✓
TAST [100]			✓	IntentDroid [45]		✓	
Griebe et al. [101]	✓			Farto et al. [46]			✓
Bielik et al. [47]			✓	MobiGUITAR [48]	✓		
AGRippin [102]	✓			Aktouf et al. [49]		✓	
THOR [103]	✓			AppAudit [50]			✓
Morgado et al. [104]	✓			Hassanshahi et al. [51]		✓	
iMPAcT [52]	✓			Deng et al. [53]			✓
Espada et al. [54]			✓	Zhang et al. [55]			✓
QUANTUM [56]	✓			CRAXDroid [57]			✓
IntentFuzzer [58]		✓		Vikomir et al. [59]			✓
Shahriar et al. [60]			✓	APSET [61]		✓	
DROIDRACER [62]			✓	EvoDroid [105]			✓
SPAG-C [106]	✓			Caiipa [107]			✓
UGA [108]		✓		AppACTS [63]			✓
CAFA [64]			✓	Holzmann et al. [109]	✓		
Guo et al. [65]		✓		Griebe et al. [66]			✓
PBGT [67]	✓			Chen et al. [110]			✓
Banerjee et al. [68]			✓	Amalfitano et al. [111]	✓		
Adinata et al. [112]			✓	A5 [69]		✓	
Suarez et al. [70]	✓			Linares et al. [71]	✓		
Sasnauskas et al. [72]		✓		AMDetector [73]			✓
RERAN [15]	✓			Yang et al. [74]	✓		
ORBIT [87]	✓			DroidTest [75]			✓
Appstrument [76]			✓	Dynodroid [89]	✓		
SPAG [113]	✓			SwiftHand [114]	✓		
A^3E [115]	✓			Avancini et al. [77]		✓	
Amalfitano et al. [116]	✓			SALES [117]			✓
LEAKDROID [78]	✓			GUIdiff [118]	✓		
Collider [119]	✓			Mirzaei et al. [18]			✓
JPF-Android [120]		✓		Mahmood et al. [79]			✓
MASHTE [121]	✓			Franke et al. [80]			✓
Dhanapal et al. [81]			✓	ACTEve [122]	✓		
SmartDroid [82]	✓			JarJarBinks [83]		✓	
TEMA [123]	✓			Sadeh et al. [124]	✓		
Hu et al. [84]	✓			A2T2 [125]	✓		
ART [126]	✓						
Count					47	12	44

automatically explore GUI bugs, where JUnit, a unit testing framework, is leveraged to automate the generation of unit testing cases.

Integration Testing. Integration testing combines all units within an app (iteratively) to test them as a group. The purpose of this phase is to infer interface defects among units or functions. It determines how efficient the units are interactive. For example, Yang et al. [58] have proposed a tool called IntentFuzzer to test the capability problems involved in inter-component communication.

System Testing. System testing is the first step that the whole app is tested as a whole. The goal of this phase is to assess whether the outlined requirements and quality standards have been fulfilled. Usually, system testing is done in a black-box style, which is usually conducted by independent testers who have no knowledge of the apps to be tested. As an example, Mao et al. [11] have proposed a testing tool named Sapienz that combines several approaches including fuzzing testing, search-based testing to systematically explore faults in Android apps.

Table 5 summarises the aforementioned test phases, where the most recurrently applied testing phase is system testing (accounting for nearly 80% of the selected publications), followed by unit testing and integration testing, respectively.

TABLE 5: Recurrent testing phases.

Tool	Unit/Regression	Integration	System	Tool	Unit/Regression	Integration	System
Zeng et al. [12]			✓	Dagger [25]			✓
Malisa et al. [26]			✓	CRASHSCOPE [27]			✓
MAMBA [28]			✓	Pretect [29]			✓
DroidMate [93]			✓	SSDA [30]			✓
TrimDroid [8]			✓	ERVA [9]			✓
Clapp et al. [10]			✓	SAPIENZ [11]			✓
RacerDroid [31]			✓	Baek et al. [94]			✓
DiagDroid [32]			✓	MobiPlay [95]			✓
MOTIF [33]			✓	DRUN [34]			✓
DroidDEV [96]			✓	GAT [35]			✓
Zhang et al. [36]			✓	Jabbarvand et al. [37]			✓
Qian et al. [38]			✓	Ermuth et al. [39]	✓		
Cadage [97]			✓	Zhang et al. [40]	✓		
Zhang et al. [41]			✓	dLens [42]			✓
Sonny et al. [98]			✓	Packeviius et al. [43]			✓
SIG-Droid [99]			✓	Knorr et al. [44]			✓
TAST [100]			✓	IntentDroid [45]		✓	
Griebe et al. [101]	✓			Farto et al. [46]			✓
Bielik et al. [47]			✓	MobiGUITAR [48]	✓		
AGRippin [102]			✓	Aktouf et al. [49]			✓
THOR [103]			✓	AppAudit [50]	✓		
Morgado et al. [104]			✓	Hassanshahi et al. [51]			✓
iMPAcT [52]			✓	Deng et al. [53]			✓
Espada et al. [54]			✓	Zhang et al. [55]		✓	
QUANTUM [56]			✓	CRAXDroid [57]	✓		✓
IntentFuzzer [58]		✓		Vikomir et al. [59]			✓
Shahriar et al. [60]	✓			APSET [61]			✓
DROIDRACER [62]			✓	EvoDroid [105]			✓
SPAG-C [106]			✓	Caiipa [107]			✓
UGA [108]			✓	AppACTS [63]			✓
CAFA [64]			✓	Holzmann et al. [109]			✓
Guo et al. [65]			✓	Griebe et al. [66]	✓	✓	
PBGT [67]			✓	Chen et al. [110]			✓
Banerjee et al. [68]			✓	Amalfitano et al. [111]	✓		
Adinata et al. [112]			✓	A5 [69]			✓
Suarez et al. [70]			✓	Linares et al. [71]	✓		
Sasnauskas et al. [72]			✓	AMDetector [73]			✓
RERAN [15]			✓	Yang et al. [74]			✓
ORBIT [87]		✓		DroidTest [75]	✓		
Appstrument [76]	✓			Dynodroid [89]			✓
SPAG [113]			✓	SwiftHand [114]			✓
A^3E [115]			✓	Avancini et al. [77]			✓
Amalfitano et al. [116]			✓	SALES [117]			✓
LEAKDROID [78]			✓	GUIdiff [118]	✓		
Collider [119]			✓	Mirzaei et al. [18]			✓
JPF-Android [120]			✓	Mahmood et al. [79]			✓
MASHTE [121]	✓	✓		Franke et al. [80]	✓		
Dhanapal et al. [81]			✓	ACTEve [122]			✓
SmartDroid [82]			✓	JarJarBinks [83]			✓
TEMA [123]	✓			Sadeh et al. [124]	✓	✓	
Hu et al. [84]	✓			A2T2 [125]	✓		
ART [126]			✓				
Count					19	7	81

Insights from RQ2 - on Test Levels

– The large majority of approaches reviewed in this SLR are about testing the whole app against given test criteria. This correlates with the test methodologies detailed below. Unit and regression testing, which would help developers assess individual functionalities in a white-box testing scenario, are limited to a few approaches.

5.3 How are the Test Approaches Built?

Our review further investigates the approaches in-depth to characterize the methodologies they leverage, the type of tests that are implemented as well as the tool support they have exploited. In this work, we refer to *test technique* as a broad concept to describe all the technical aspects related to testing, while we constrain the term *test methodology* to

specifically describe the concrete methodology that a test approach applies.

5.3.1 Test methodologies

Table 6 enumerates all the testing methodologies we observed in our examination.

Model-based Testing is a testing methodology that goes one step further than traditional methodologies by automatically generating test cases based on a model, which describes the functionality of the system under test. Although such methodology incurs a substantial, usually manual, effort to design and build the model, the eventual test approach is often extensive, since test cases can be automatically generated and executed. Our review has revealed that model-based testing is the most common methodology used in Android testing literature: 63% of publications involve some model-based testing steps. Takala et al. [123] present a comprehensive documentation on their experiences in applying a model-based GUI testing to Android apps. They typically discuss how model-based testing and test automation are implemented, how apps are modelled, as well as how tests are designed and executed.

Search-based Testing is using the metaheuristic search techniques to generate software tests [129], with the aim to detect as many bugs as possible, especially the most critical ones, in the system under test. In [105], the authors developed an evolutionary testing framework for Android apps. Evolutionary testing is a form of search-based testing, where an individual corresponds to a test case, and a population comprised of many individuals is evolved according to certain heuristics to maximize the code coverage. Their technique thus tackles the common shortcoming of using evolutionary techniques for system testing. In order to generate the test suites in an effective and efficient way, Amalfitano et al. [102] proposed a novel search-based testing technique based on the combination of genetic and hill climbing techniques.

Random Testing is a software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is a *pass* or a *fail* [130]. In the absence of specifications, program exceptions are used to detect test case *fails*. Random testing is also acquired by almost all other test suite generation methodologies and serves as a fundamental technique. Random testing has been used in several literature works [89], [97], [103], [108], [126].

Fuzzing Testing is a testing technique that applies invalid, unexpected, or random data as inputs to a testing object. It is commonly used to test for security problems in software or computer systems. The main focus then shifts to monitoring the program for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. A number of research papers (e.g., [23], [84]) have explored this type of testing via automated or semi-automated *fuzzing*. Fuzzing testing is slightly different from random testing, as it mainly embraces, usually on purpose, unexpected, invalid inputs and focuses on monitoring crashes/exceptions of the tested apps while random testing does not need to conform to any of such software specifications.

A/B Testing provides a means for comparing two variants of a testing object, and hence determining which of the two variants is more effective. A/B testing is recurrently used for statistical hypothesis tests. In [112], Adinata et al. have applied A/B testing to test mobile apps, where they have solved three challenges of applying A/B testing, including element composition, variant delivery and internet connection. Holzmann et al. [109] conduct A/B testing through a multivariate testing tool.

Concolic Testing is a hybrid software verification technique that performs symbolic execution, a classical technique which treats program variables as symbolic variables, along with a concrete execution path (testing on particular inputs). Anand et al. [122] propose a concolic testing technique, CONTEST, to alleviate the path explosion problem. They develop a concolic-testing algorithm to generate sequences of events. Checking the subsumption condition between event sequences allows the algorithm to trim redundant event sequences, thereby, alleviating path explosion.

Mutation Testing is used to evaluate the quality of existing software tests. It is performed by selecting a set of mutation operators and then applying them to the source program, one operator at a time, for each relevant program location. The result of applying one mutation operator to the program is called a mutant. If the test suite is able to detect the change (i.e., one of the tests fails), then the mutant is said to be killed. In order to realize an end-to-end system testing of Android apps in a systematic manner, Mahmood et al. [105] propose EvoDroid, an evolutionary approach of system testing of apps, in which two types of mutation (namely, input genes and event genes) are leveraged to identify a set of test cases that maximize code coverage. Mutation testing-based approaches are however not common in the Android literature.

Overall, our review has shown that the literature often combines several methodologies to improve test effectiveness. In [108], the authors combined model-based testing with random testing to complete the testing. Finally, EvoDroid [105] is a framework that explores model-based, search-based and mutation testing techniques.

5.3.2 Test types

In general, there are three types of testing, namely the *White-box testing*, *Black-box testing*, and *Grey-box testing*. Table 7 summarizes these testing types by emphasizing on the ideal tester (the software developer or a third-party), on whether knowledge on implementation details is fully/partially/not required.

White-box testing is a scenario in which the software is examined based on the knowledge of its implementation details. It is usually applied by the software developers in early development stages when performing *unit testing*. Another common usage scenario is to perform thorough tests once all software components are assembled (known as *regression testing*). In this SLR, when an approach requires app source (or byte) code knowledge, whether obtained directly or via reverse engineering, we consider it a white-box approach.

Black-box testing, on the other hand, is a scenario where internal design/implementation of the tested object is not

TABLE 6: Test method employed in the literature.

Tool	Model-based	Search-based	Random	Fuzzing	A/B	Concolic	Mutation	Tool	Model-based	Search-based	Random	Fuzzing	A/B	Concolic	Mutation
Zeng et al. [12]			✓					Dagger [25]	✓						
Malisa et al. [26]	✓							CRASHSCOPE [27]	✓						
MAMBA [28]	✓							DroidMate [93]	✓						
SSDA [30]	✓			✓				TrimDroid [8]	✓						
ERVA [9]	✓							Clapp et al. [10]	✓						
SAPIENZ [11]		✓		✓				RacerDroid [31]	✓						
Baek et al. [94]	✓							DiagDroid [32]			✓				
MOTIF [33]	✓							DRUN [34]	✓						
DroidDEV [96]	✓							GAT [35]	✓						
Zhang et al. [36]	✓							Jabbarvand et al. [37]	✓						
Qian et al. [38]	✓							Ermuth et al. [39]	✓						
Cadage [97]	✓							Zhang et al. [40]	✓						
Zhang et al. [41]	✓							dLens [42]	✓						
Sonny et al. [98]	✓							Packeviius et al. [43]	✓						
SIG-Droid [99]	✓							TAST [100]	✓						
IntentDroid [45]				✓				Farto et al. [46]	✓						
Bielik et al. [47]	✓							MobiGUITAR [48]	✓						
AGRippin [102]		✓					✓	Aktouf et al. [49]	✓						
THOR [103]			✓					AppAudit [50]	✓						
Morgado et al. [104]	✓							Hassanshahi et al. [51]	✓			✓			
iMPAcT [52]	✓							Deng et al. [53]							✓
Espada et al. [54]	✓							Zhang et al. [55]			✓				
QUANTUM [56]	✓							CRAXDroid [57]				✓			
IntentFuzzer [58]				✓				Shahriar et al. [60]	✓			✓			
APSET [61]	✓							DROIDRACER [62]	✓						
EvoDroid [105]	✓	✓					✓	SPAG-C [106]	✓						
Caiipa [107]				✓				UGA [108]	✓		✓				
AppACTS [63]			✓					Holzmann et al. [109]					✓		
Guo et al. [65]	✓							Griebe et al. [66]	✓						
PBGT [67]	✓							Amalfitano et al. [111]	✓						
Adinata et al. [112]					✓			A5 [69]			✓				
Suarez et al. [70]	✓							Linares et al. [71]	✓						
Sasnauskas et al. [72]	✓			✓				AMDetector [73]	✓						
RERAN [15]	✓							Yang et al. [74]	✓						
ORBIT [87]	✓							Dynodroid [89]			✓				
SwiftHand [114]	✓							A^{3}E [115]	✓						
Avancini et al. [77]	✓							SALES [117]	✓						
LEAKDROID [78]	✓							GUIdiff [118]			✓				
Collider [119]	✓					✓		JPF-Android [120]	✓						
Mahmood et al. [79]	✓			✓				ACTEve [122]						✓	
SmartDroid [82]	✓							JarJarBinks [83]				✓			
TEMA [123]	✓							Hu et al. [84]			✓				
A2T2 [125]	✓							ART [126]			✓				
				Count					65	3	11	11	2	2	3

TABLE 7: Common test types.

Testing Type	Ideal Tester	Implementation Knowledge
White-box	Developer	Known
Black-box	Independent Tester	Unknown
Grey-box	Independent Tester	Partially Known

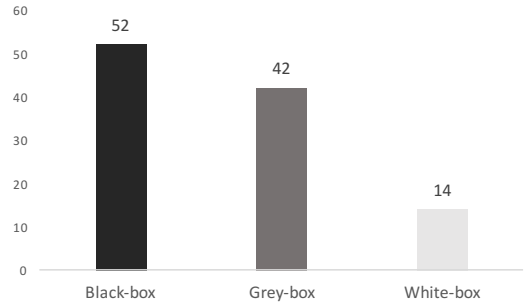


Fig. 7: Breakdown of examined publications regarding their applied testing types.

required. Black-box testing is often conducted by third-party testers, who have no relationships with the developers of tested objects. If an Android app testing process only requires the installation of the targeted app, we reasonably put it under this category.

Grey-box testing is a trade-off between white-box testing and black-box. It does not require the testers to have full knowledge on the source code where white-box testing needs. Instead, it only needs the testers to know some limited specifications like how the system components interact. For the investigations of our SLR, if a testing approach requires to extract some knowledge (e.g., from the Android manifest configuration) to guide its tests, we consider it a grey-box testing approach.

Fig. 7 illustrates the distribution of test types applied by examined testing approaches. White-box testing is the least used type, far behind black-box and grey-box testing. This is expected because Android apps are usually compiled and distributed in APK format, so testers in most scenarios have no access to source code. We also wish to address that one literature can make use of more than one testing type, this is why the sum of the three types in Fig. 7 is larger than 103.

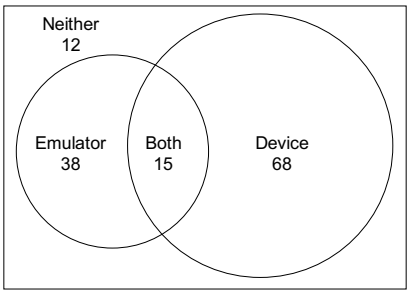


Fig. 8: Venn Diagram of Testing Environment.

5.3.3 Test environments

Unlike static analysis of Android apps [22], testing requires to actually run apps on an execution environment such as a *real device* or an *emulator*.

Real Device has a number of advantages: they can be used to test apps w.r.t compatibility aspects [41], [59], [63], energy consumption [42], [68], [71], and the poor responsiveness issue [29], [74]. Unfortunately, using real devices is not efficient, since it cannot scale in terms of execution time and resources (several devices may be required).

Emulator, on the contrary, can be scalable. When deployed on the cloud, using the emulator can grant a tester great computing resources and carry out parallel tests at a very large scale [79]. Unfortunately, emulators are ineffective for security-relevant tests, since some malware have the functionality to detect whether they are running on an emulator. If so, they may decide to refrain from exposing their malicious intention [131]. Emulators also introduce huge overhead when mimicking real-life sensor inputs, e.g., requiring altering the apps under testing at source code level [101].

Emulator + Real Device, can be leveraged together to test Android apps. For example, one can first use an emulator to launch large-scale app testing for pre-selecting a subset of relevant apps and then resort to real devices for more accurate testing.

As can be seen from Figure 8, real devices are largely used by 68 publications in our final list. Only 38 publications used emulators, despite the fact that they are cheap. 15 publications chose both environments to avoid disadvantages of either. Deducting these 15 publications, we can calculate that 23 publications focused solely on emulators, where 53 publications selected real devices as the only environment.

5.3.4 Tool support

While performing the SLR, we have observed that several publicly available tools were recurrently leveraged to implement or complement the state-of-the-art approaches. Table 8 enumerates such tools with example references to works where they are explored.

AndroidRipper is a tool for automatic GUI testing of Android apps. It is driven by a user-interface ripper that automatically and systematically travels the app's GUI aiming at exercising a given app in a structured way. In order to generate test cases in an effective and efficient way, Amalfitano et al. [102] extend this work with search-based testing techniques, where genetic and hill climbing algorithms are considered.

EMMA is an open-source toolkit for measuring and reporting Java code coverage. Since Android apps are written in Java, researchers often use EMMA to compute the code coverage of their Android app testing approaches, including EvoDroid [105] and SIG-Droid [99].

Monkey is a test framework released and maintained by Google, the official maintainer of Android. It generates and sends pseudo-random streams of user/system events into the running system. This functionality is exploited in the literature to automatically identify defects of ill-designed apps. As an example, Hu et al. [84] leveraged Monkey to identify GUI bugs of Android apps. The randomly generated test cases (events) are fed into a customized Android system that produces log/trace files during the test. Those log/trace files can then be leveraged to perform post analysis and thereby to discover event-related bugs.

RERAN is a record and replay tool for testing Android apps. Unlike traditional record-and-reply tools, which are inadequate for Android apps because of their expressiveness on smartphone features, RERAN supports sophisticated GUI gestures and complex sensor events. Moreover, RERAN achieves accurate timing requirements among various input events. A^3E [115] for example uses RERAN to record its targeted and depth-first exploration for systematic testing of Android apps. Those recorded explorations can later be replayed so that to benefit debuggers in quickly localizing the exact event stream that has led to the crash.

Robotium is an open-source test framework, which has full support for native and hybrid apps. It also eases the way to write powerful and robust automatic black-box UI tests of android apps. SIG-Droid [99] for example leverages Robotium to execute its generated test cases (with the help of symbolic execution). We have found during our review that Robotium were most frequently leveraged by state-of-the-art testing approaches.

Robolectric is a unit testing framework, which simulates the Android execution environment (either on a real device or on an emulator) in a pure Java environment. The main advantage of doing that is to improve the testing efficiency because tests running inside a JVM are much faster than that of running on an Android device (or even emulator), where it usually takes minutes to build, deploy and launch an app. Sadeh et al. [124] have effectively used Robolectric framework to conduct unit testing for their calculator application. They have found that it is rather easy to write test cases with this framework, which requires only a few extra steps and abstractions. Because testers do not need to maintain a set of fake objects and interfaces, it is even preferable for complex apps.

Sikuli uses visual technology to automate GUI testing through screenshot images. It is particularly useful when there is no easy way to obtain the app source code or the internal structure of graphic interfaces. Lin et al. [106], [113] leveraged Sikuli in their work to enable record-and-replay testing of Android apps, where the user interactions are saved beforehand in Sikuli test formats (as screenshot images).

TABLE 8: Summary of basic tools that are frequently leveraged by other testing approaches.

Tool	Brief Discription	Example Usages
AndroidRipper	An automated GUI-based testing tool	Yang et al. [74], Amalfitano et al. [111], [116], AGRippin [102], MobiGUITAR [48]
EMMA	A free Java code coverage measuring tool	Mirzaei et al. [18], Mahmood et al. [79], SIG-Droid [99], BBOXTESTER [23], EvoDroid [105]
Monkey	An automated testing tool that generates and executes randomly generated test cases	Hu et al. [84], BBOXTESTER [23], TAST [100],
RERAN	A timing- and touch-sensitive record and replay tool for Android apps	UGA [108], dLens [42], A^{3}E [115]
Robotium	An open-source test framework for writing automatic black box testing cases for Android apps	A2T2 [125],Chen et al. [110], UGA [108], THOR [103], Yang et al. [74], ORBIT [87], Mahmood et al. [79], AGRippin [102], Guo et al. [65], SIG-Droid [99]
Robolectric	A unit test framework that enables tests run inside JVM instead of DVM	Sadeh et al. [124], Mirzaei et al. [18]
Sikuli	A visual technology to automate and test GUIs using screenshot images	SPAG [113], SPAG-C [106]

Insights from RQ3 - on Used Techniques

– Given the complexity of interactions among components in Android apps as well as with the operating system, it is not surprising that most approaches in the literature resort to "model-based" techniques which build models for capturing the overall structure and behaviour of apps to facilitate testing activities (e.g., input generation, execution scenarios selection, etc.).
– The unavailability of source code for market apps make white-box techniques less attractive than grey-box and black-box testing for assessing apps in the wild. Nevertheless, our SLR shows that the research community has not sufficiently explored testing approaches that would directly benefit app developers during the development phase.
– Tool support for building testing approaches is abundant. The use of the Robotium open source test framework by numerous approaches once again demonstrates the importance of making tools available to stimulate research.

5.4 To What Extent are the Approaches Validated?

Several aspects must be considered when assessing the effectiveness of a testing approach. We consider in this SLR the measurements performed on *code coverage* as well as on *accuracy*. We also investigate the use of a *ground truth* to validate performance scores, the *size of the experimental dataset*.

Coverage is a key aspect for estimating how well the program is tested. Larger coverage generally correlates with higher possibilities of exposing potential bugs and vulnerabilities, as well as uncovering malicious behaviour. There are numerous coverage metrics leveraged by state-of-the-art works. For example, for evaluating **Code Coverage**, metrics such as *LoC* (Lines of Code) [11], [102], [105], *Block* [97], Method [108], [115], Branch [114] have been proposed in our community. In order to profile the **Accuracy** of testing approaches, other coverage metrics are also proposed in the literature such as bugs [42] and vulnerabilities [45] (e.g., *how many known vulnerabilities can the evaluated testing approach cover?*). Table 9 enumerates the coverage metrics used in the literature, where *LoC* appears to be the most concerned metric.

TABLE 9: Assessment Metrics (e.g., for Coverage, Accuracy).

Metrics (# of)	Example Publications
LoC	EvoDroid [105], AGRippin [102], THOR [103] Zeng et al. [12], SAPIENZ [11]
Block	Cadage [97]
Branch	SwiftHand [114]
Method	UGA [108], A^{3}E [115]
Exception	Zhang et al. [55]
Action	ORBIT [87]
Activity	A^{3}E [115], Avancini et al. [77], Malisa et al. [26] MAMBA [28], Clapp et al. [10]
Service	Zhang et al. [40]
Bug	dLens [42], TEMA [123], Hu et al. [84], MobiGUITAR [48]
Defect	APSET [61]
Fault	QUANTUM [56], Vikomir et al. [59], Sonny et al. [98]
Crash	Shahriar et al. [60], Caiipa [107], .CRASHSCOPE [27]
Vulnerability	Sadeh et al. [124], IntentDroid [45]
Leakage	CRAXDroid [57], Yang et al. [58]

Ground Truth refers to a reference dataset where each element is labelled. In this SLR, we consider two types of ground truths. The first is related to malware detection approaches: the ground truth then contains apps labelled as benign or malicious. As an example, the Drebin [132] dataset has recurrently been leveraged as ground truth to evaluate testing approaches [133]. The second is related to vulnerability and bug detection: the ground truth represents code that is flagged to be vulnerable or buggy based on the observation of bug reports summited by end users or bug fix histories committed by developers [55], [84].

Dataset Size The *Dataset Size* is the number of apps tested in the experimental phase. We can see from Fig. 9 that most works (ignoring outliers) carried out experiments on no more than 100 apps, with a median number of 8 apps. Comparing to the distribution of the number of evaluated apps summarized in an SLR of static analysis of Android apps [22], where the median and maximum numbers are respectively 374 and 318,515, far bigger than the number of apps considered by testing approaches. This result is somehow expected as testing approaches (or dynamic analysis approaches) are generally not scalable.

Insights from RQ4 - on Approach Validation

Although literature works always provide evaluation section to provide evidence (often through comparison) that their approaches are effective, their reproducibility is still challenged by the fact that there is a lack of estab-

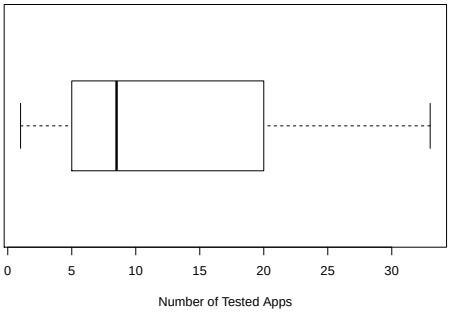


Fig. 9: The distribution of the number of tested apps (outliers are removed).

lished ground truth and benchmarks. Yet, reproducibility is essential to ensure that the field is indeed progressing based on a baseline performance, instead of relying on subjective observation by authors and on datasets with variable characteristics.

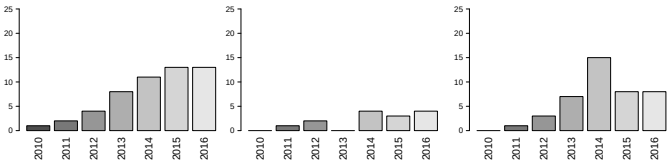
6 Discussion

Research on Android app testing has been prolific in the past years. Our discussion will focus on the trends that we observed while performing this SLR, as well as on the challenges that the community should still attempt to address.

6.1 Trend Analysis

The development of the different branches in the taxonomy is disparate.

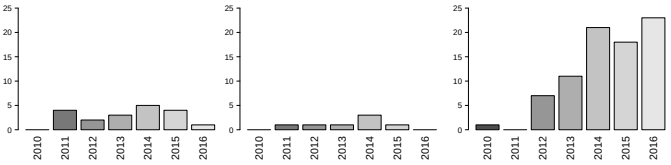
Fig. 10 illustrates the trend in testing types over the years. Together, black-box and grey-box testing are involved in 90% of the research works. Their evolution is thus reflected by the overall evolution of research publications (cf. Fig. 4). White-box testing remains low in all years.



(a) Black-box. (b) White-box. (c) Grey-box.

Fig. 10: Trend of Testing Types.

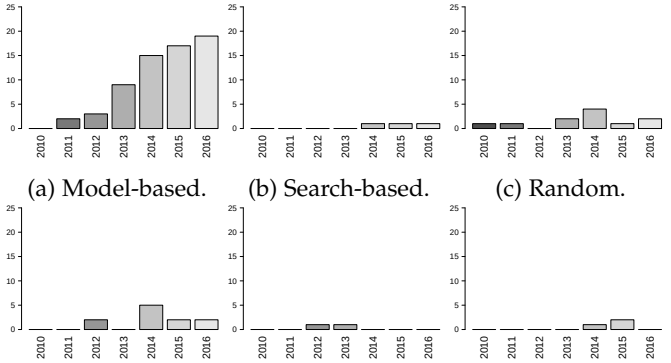
Fig. 11 presents the evolution over time of works addressing different test levels. Unit/regression and integration testing phases include a low, but stable, number of works every year. Overall, system testing has been heavily used in the literature and has even doubled between 2012 and 2014. System testing of Android apps is favored since app execution is done on a specific virtual machine environment with numerous runtime dependencies: it is not straightforward to isolate a single block for unit/regression testing or to test the integration of two components without interference from other components. Nevertheless, with the increasing use of code instrumentation [14], there are new opportunities to eventually slice android apps for performing more grey-box and white-box testing.



(a) Unit/Regression. (b) Integration. (c) System.

Fig. 11: Trend of Testing Levels.

Trend analysis of testing methods in Fig. 12 confirms that model-based testing is dominating in the literature of Android app testing, and its evolution is reflected in the overall evolution of testing approaches. Most approaches indeed start by constructing a GUI model or a call graph (CG) to generate efficient test cases. In the last couple of years, mutation testing has been appearing in the literature, similarly to search-based techniques.



(a) Model-based. (b) Search-based. (c) Random.

(d) Fuzzing. (e) Concolic. (f) Mutation.

Fig. 12: Trend of Testing Methods.

With regard to testing targets, Fig. 13(a-b) shows that the graphical user interfaces, as well as the event mechanism, are continuously at the core of research approaches. Since Android *Activities* (i.e., the UIs) are the main entry points for executing test cases, the community will likely continue to develop black-box and grey-box test strategies that increase interactions with GUI to improve code coverage. Inter-component and inter-application communications, on the other hand, have been popular targets around 2014.

With regard to testing objectives, Fig. 13(c-h) shows that *security* concerns have attracted a significant amount of research, although the output has been decreasing in the last couple of years. Bug/defect identification, however, has somewhat stabilized.

6.2 Evaluation of Authors

Android testing is a new field of research which has attracted several contributions over the years due to the multiple opportunities that it offers for researchers to apply theoretical advances in the domain of software testing. We emphasize the attractiveness of the field by showing in Fig. 14 the evolution of single authors contributing to research approaches. We count in each year, the *Total Authors*

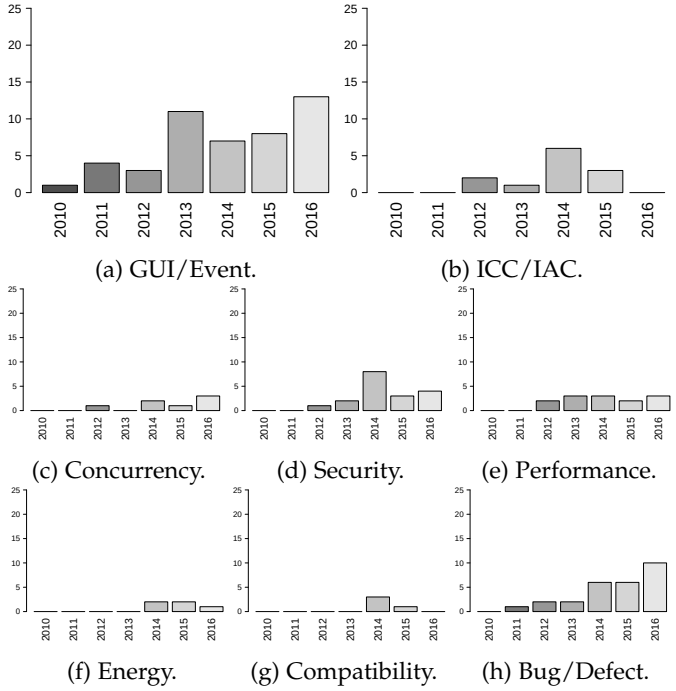


Fig. 13: Trend of Testing Targets and Objectives.

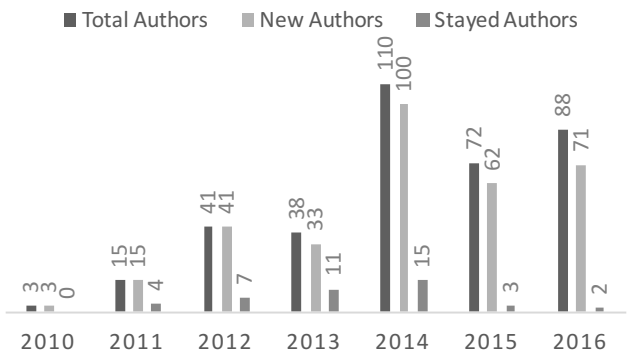


Fig. 14: Trend in community authors. "New Authors" and "Stayed Authors" indicate the number of authors that enter the field (no relevant publications before) and have stayed in the field (they will keep publishing in the following years).

who participated in at least one of our selected publications, the *New Authors* that had had no selected publication until that year, and the number of *Stayed Authors* who had publications selected both that year and the years to come. Overall, the figures raise several interesting findings:

- Every year, the community of Android testing research authors is almost entirely renewed.
- Only a limited number of researchers publish again in the theme after one publication.

These facts may suggest that the research in Android app testing is often governed by opportunities. Furthermore, challenges (e.g., building a sound GUI event model) quickly arise, making authors lose interest in pursuing in this research direction. Although we believe that the fact that the topic is within reach of a variety of authors from other backgrounds is good for bringing new ideas and cross-fertilizing, the maturity of the field will require commitment from more authors staying in the field.

6.3 Research Output Usability

In the course of our investigations for performing the review, we have found that the research community on Android app testing seldom contributes with reusable tools (e.g., implementation of approaches for GUI testing), not even mention to contribute with open source testing tools. Yet, the availability of such tools is necessary not only to limit the efforts in subsequent works but also to encourage true progress beyond the state-of-the-art.

Despite most testing approaches are not made publicly available, it is nevertheless gratifying to observe that some of them have been leveraged in industry. For example, research tool TEMA has now been integrated into the RATA project[16], where researchers (from Tampere University of Technology) and practitioners (from Intel Finland, OptoFidelity, and VTT) work together to provide robot-assisted test automation for mobile apps. Another research tool named SAPIENZ has led to a start-up called MaJiCKe and recently been acquired by Facebook London, being the core component of Facebook's testing solutions for mobile apps.

16. http://wiki.tut.fi/RATA/WebHome

6.4 Open Issues and Future Challenges

Although the publications we chose all have their own solid contributions, some authors posed open issues and future challenges to call in more research attention to the domain. We managed to collect the concerns and summarized as follows:

- **Satisfying Fastidious Pre-conditions.** One recurrently discussed issue is to generate test cases that can appropriately satisfy pre-conditions such as login to an app. When the oracles generate events to traverse the activities of Android apps, some particular activities are extremely hard to be touched. A publicly known condition is to tap the same button for 7 consecutive times in order to trigger developer mode [12], [99]. Another example would be to break through the login page which requires a particular combination of user account and passwords. Both preconditions are clearly not easy to be satisfied during the process of testing Android apps.
- **Modelling Complex Events (e.g., Gestures or Non-user Events).** In addition to simple events such as clicking, Android OS also involves quite a lot of complex events such as user gestures (swipe, long press, zoom in/out, spin, etc.) and system events (network connectivity, events coming from light, pressure and temperature sensors, GPS, fingerprint recognizer, etc.). All the events would introduce non-deterministic behaviours if they are not properly modelled. Unfortunately, at the moment, most of our reviewed papers only tackle simple events like clicking, letting other events remain untouched [67], [101].
- **Bridging Incompatible Instruction Sets.** To improve the performance of Android apps, Google provides a toolset, i.e., the Android Native Developer Kit

(NDK), allowing app developers to implement time-intensive tasks via C/C++. Those tasks implemented with C/C++ are closely dependent on the CPU instruction sets (e.g., Intel or ARM) and hence can only be launched in right instruction sets, e.g., tasks implemented based on the ARM architecture can only be executed on ARM-based devices). However, as most mobile devices nowadays are assembled with ARM chips while most PCs running Android emulators are assembled with Intel chips, running ARM-based emulators on Intel-based PCs are extremely slow, this gap has caused problems for emulator-based testing approaches [95].

- **Evaluating Testing Approaches Fairly.** Frequently, researchers complain about the fact that our community has not provided a reliable coverage estimator to approximate the coverage (e.g., code coverage) of testing approaches and to fairly compare them [12], [29], [41], [43]. Although some outstanding progress has been made for developing estimation tools [23], our SLR still indicates that there does not exist any universally acquired tool that supports fair comparison among testing approaches. We, therefore, urge our fellow researchers to appropriately resolve this open issue and subsequently contribute to our community a reliable artefact benefiting many aspects of future research studies.
- **Addressing Usability Defect.** The majority of the research studies focuses on functional defects of Android apps. The usability defect does not attract as much attention as the users are concerned [53]. Usability defect like poor responsiveness [74] is a major drawback of Android apps and receives massive complaints from users. Bad view organization on the screen arising from incompatibility and repetitive imprecise recognition of user gestures also imply bad user experience.

6.5 New Research Directions

In light of the SLR summary of the state-of-the-art and considering the new challenges reported in the literature, there are opportunities for exploring new testing applications to improve the quality of Android apps or/and increase confidence in using them safely. We now enumerate three example directions:

6.5.1 Validation of app updates

Android app developers regularly update their apps for various reasons, including keeping them attractive to the user base[17]. Unfortunately, recent studies [134] have shown that updates of Android apps often come with more security vulnerabilities and functional defects. In this context, the community could investigate and adapt regression techniques for identifying defect-prone or unsafe updates. To accelerate the identification of such issues in updates, one can consider exploring approaches with behavioural equivalence, e.g., using "record and replay" test-case generation techniques.

17. https://savvyapps.com/blog/how-often-should-you-update-your-app

6.5.2 Accounting for the ecosystem fragmentation

As previously highlighted, the fragmentation of the Android ecosystem (with a high variety in operating system versions where a given app will be running, as well as a diversity of hardware specifications) is a serious challenge for performing tests that can expose all issues that a user might encounter on his specific device runtime environment. There is still room to investigate test optimization and prioritization for Android to cover a majority of devices and operating system versions. For example, on top of modelling apps, researchers could consider modelling the framework (and its variabilities) and account for it during test execution.

6.5.3 Code prioritization vs test prioritization

Finally, we note that Android apps are becoming larger and larger in terms of size, including obsolete code for functionalities that are no longer needed, or to account for the diversity of devices (and their OS versions). For example, in large companies, because of developer rotation, "dead" code/functionality may remain hidden in plain sight of app code without development teams risking to remove them. As a result, the effort thrown in maintaining those apps increases continuously, where consequently the testing efforts required to verify the functional correctness of those apps also boost. Therefore, to alleviate this problem, we argue that testing such apps clearly necessitates optimizing the selection of code that must be tested in priority. Test cases prioritization must then be performed in conjunction with a code optimization process to focus on actively used code w.r.t. user interactions to the app.

7 Threats to Validity

We have identified the following threats to validity in our study:

On potential misses of literature – We have not considered for our review books and Master or PhD dissertations related to the Android testing. This threat is mitigated by the fact that the content of such publications is eventually presented in peer-reviewed venues which we have considered. We have also considered only publications written in English. Nevertheless, while searching with the compiled English keywords, we have also found a few papers written in other languages, such as German and Chinese. The number of such non-English papers remain however significantly small, compared with the collected English literature, suggesting that our SLR is likely complete. Last but not the least, although we have refined our searching keywords several times, it is still possible that some synonyms are missed in this work. To mitigate this, we believe that natural language processing (NLP) could be leveraged to disclose such synonyms. We, therefore, consider it as our future work towards engineering sound keywords for supporting SLR.

On data extraction errors – Given that papers are often imprecise with information related the aspects that we have investigated, the extracted data may not have been equally reliable for all approaches, and data aggregation can still include several errors as warned by Turner et al. [135] for such studies. We have nevertheless strived to mitigate this issue by applying a cross-checking mechanism on the

extracted results, following the suggestion of Brereton et al. [20]. To further alleviate this, we plan to validate our extracted results through their original authors.

On the representativeness of data sources and metrics – We have implemented the "major venues search" based on the venue ranking provided by the CCF. This ranking is not only potentially biased towards a specific community of researchers but may also change from one year to another. A replication of this study based on other rankings may lead to different primary publications set, although the overall findings will likely remain the same since most major venues continue to be so across years and across ranking systems.

The aspects and metrics investigated in this approach may also not be exhaustive or representative of everything that characterizes testing. Nevertheless, these metrics have been collected from testing literature to build the taxonomy and are essential for comparing approaches.

8 Related Work

Mobile operating systems, in particular, the open-source Android platform, have been fertile ground for research in software engineering and security. Several surveys and reviews have been performed on approaches for securing [136], [137], or statically analysing Android apps [22]. A systematic literature review is indeed important to analyse the contributions of a community to resolve the challenges of a specific topic. In the case of Android testing, such a review is missing.

Several works in the literature have however attempted to provide an overview of the field via surveys or general systematic mappings on mobile application testing techniques. For example, the systematic mapping of Sein et al. [138] addresses all together Android, iOS, Symbian, Silverlight and Windows. The authors have provided a higher-level categorization of techniques into five groups: 1) usability testing; 2) test automation; 3) context-awareness; 4) security and 5) general category. Méndez-Porras et al. [139] have provided another mapping, focusing on a more narrowed field, namely automated testing of mobile apps. They discuss two major challenges for automating the testing process of mobile apps, which are an appropriate set of test cases and an appropriate set of devices to perform the testing. Our work, with this SLR, goes in-depth to cover different technical aspects of the literature on specifically Android app testing (as well as test objectives, targets and publication venues).

Other related works have discussed directly the challenges of testing Android apps in general. For example, Amalfitano et al. [140] analyse specifically the challenges and open issues of testing Android apps, where they have summarized suitable and effective principles, guidelines, models, techniques and technologies related to testing Android apps. They enumerate existing tools and frameworks for automated testing of Android apps. They typically summarize the issues of software testing regarding non-functional requirements, including performance, stress, security, compatibility, usability, accessibility, etc.

Gao et al. [141] present a study on mobile testing-as-a-service (MTaaS), where they discuss the basic concepts of performing MTaaS. Besides, the motivations, distinct features, requirements, test environments and existing approaches are also discussed. Moreover, they have also discussed the current issues, needs and challenges of applying MTaaS in practice.

More recently, Starov et al. [142] performed a state-of-the-art survey to look into a set of cloud services for mobile testing. Based on their investigation, they divide the cloud services of mobile testing into three sub-categories: 1) Device clouds (mobile cloud platforms); 2) Services to support application lifecycle management and 3) Tools to provide processing according to some testing techniques. They also argue that it is essential to migrate the testing process to the clouds, which would make teamwork become possible. Besides, it can also reduce the testing time and development costs.

Muccini et al. [143] conducted a short study on the challenges and future research directions for testing mobile apps. Based on their study, they find that (1) Mobile apps are so different from traditional ones and thus they require different and specialized techniques in order to test them and (2) There seems to have many challenges. As an example, the performance, security, reliability and energy are strongly affected by the variability of the testing environment.

Janicki et al. [144] surveyed the obstacles and opportunities in deploying model-based GUI testing of mobile apps. Unlike conventional automatic test execution, model-based testing goes one step further by considering the automation of test generation phases as well. Based on their studies, they claim that the most valuable kind of research need (as future work) is to perform a comparative experiment on using conventional test and model-based automation, as well as exploratory and script-based manual testing to evaluate concurrently on the same system and thus to measure the success of those approaches.

Finally, the literature includes several surveys [136], [145]–[147] on Android, which cover some aspects of Android testing. As an example, Tam et al. [136] have studied the evolution of Android malware and Android analysis techniques, where various Android-based testing approaches such as A^3E have been discussed.

9 Conclusion

We report in this paper on a systematic literature review performed on the topic of Android app testing. Our review has explored 103 papers that were published in major conferences, workshops and journals in software engineering, programming language, and security domain. We have then proposed a taxonomy of the related research exploring several dimensions including the objectives (i.e., what functional or non-functional concerns are addressed by the approaches) that were pursued, and the techniques (i.e., what type of testing methods – mutation, concolic, etc.) that were leveraged. We have further explored the assessments presented in the literature, highlighting the lack of established benchmarks to clearly monitor the progress made in the field. Finally, beyond quantitative summaries, we have provided a discussion on future challenges and proposed new research directions of Android testing research for further ensuring the quality of apps with regards to compatibility issues, vulnerability-inducing updates, etc.

APPENDIX

The full list of examined primary publications are enumerated in Table A1.

REFERENCES

[1] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *The 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, 2016.

[2] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017.

[3] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. Understanding the test automation culture of app developers. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[4] Li Li. Mining androzoo: A retrospect. In *The Doctoral Symposium of 33rd International Conference on Software Maintenance and Evolution (ICSME-DS 2017)*, 2017.

[5] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[6] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[7] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[8] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in gui testing of android applications. In *International Conference on Software Engineering*, 2016.

[9] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. Automatically verifying and reproducing event-based races in android apps. In *International Symposium on Software Testing and Analysis*, 2016.

[10] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing gui event traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[11] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *International Symposium on Software Testing and Analysis*, 2016.

[12] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. Automated test input generation for android: are we really there yet in an industrial case? In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[13] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: Automated ad fraud detection for android apps. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*, 2018.

[14] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.

[15] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *International Conference on Software Engineering*, 2013.

[16] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018.

[17] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 226–237, 2016.

[18] Nariman Mirzaei, Sam Malek, Corina S. Psreanu, Naeem Esfahani, and Riyadh Mahmood. Testing android apps through symbolic execution. In *ACM SIGSOFT Software Engineering Notes*, 2012.

[19] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. In *Technical report, EBSE Technical Report EBSE-2007-01*. sn, 2007.

[20] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[21] Phu H Nguyen, Max Kramer, Jacques Klein, and Yves Le Traon. An extensive systematic review on the model-driven development of secure systems. *Information and Software Technology*, 68:62–81, 2015.

[22] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[23] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards black box testing of android apps. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 501–510. IEEE, 2015.

[24] Chao-Chun Yeh and Shih-Kun Huang. Covdroid: A black-box testing coverage system for android. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 3, pages 447–452. IEEE, 2015.

[25] Chao Yang, Guangliang Yang, Ashish Gehani, Vinod Yegneswaran, Dawood Tariq, and Guofei Gu. Using provenance patterns to vet sensitive behaviors in android apps. In *ACM SIGSAC conference on Computer and communications security*, 2016.

[26] Luka Malisa, Kari Kostiainen, Michael Och, and Srdjan Capkun. Mobile application impersonation detection using dynamic user interface extraction. In *European Symposium on Research in Computer Security*, 2016.

[27] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *IEEE International Conference on Software Testing, Verification and Validation*, 2016.

[28] Joseph Chan Joo Keng, Lingxiao Jiang, Tan Kiat Wee, and Rajesh Krishna Balan. Graph-aided directed testing of android applications for checking runtime privacy behaviours. In *International Workshop on Automation of Software Test*, 2016.

[29] Yu Kang, Yangfan Zhou, Min Gao, Yixia Sun, and Michael R Lyu. Experience report: Detecting poor-responsive ui in android applications. In *International Symposium on Software Reliability Engineering*, 2016.

[30] Yongjian Hu and Iulian Neamtiu. Fuzzy and cross-app replay for smartphone apps. In *International Workshop on Automation of Software Test*, 2016.

[31] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. Generating test cases to expose concurrency bugs in android applications. In *International Conference on Automated Software Engineering*, 2016.

[32] Yu Kang, Yangfan Zhou, Hui Xu, and Michael R Lyu. Diagdroid: Android performance diagnosis via anatomizing asynchronous executions. In *International Conference on Foundations of Software Engineering*, 2016.

[33] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *International Conference on Mobile Software Engineering and Systems*, 2016.

[34] Quan Sun, Lei Xu, Lin Chen, and Weifeng Zhang. Replaying harmful data races in android apps. In *International Symposium on Software Reliability Engineering Workshop*, 2016.

[35] Xiangyu Wu, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. Testing android apps via guided gesture event generation. In *Asia-Pacific Software Engineering Conference*, 2016.

[36] Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated test generation for detection of leaks in android applications. In *International Workshop in Automation of Software Test*, 2016.

[37] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Energy-aware test-suite minimization for android apps. In *International Symposium on Software Testing and Analysis*, 2016.

[38] Ju Qian and Di Zhou. Prioritizing test cases for memory leaks in android applications. In *Journal of Computer Science and Technology*, 2016.

[39] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 82–93. ACM, 2016.

[40] Tao Zhang, Jerry Gao, Oum-El-Kheir Aktouf, and Tadahiro Uehara. Test model and coverage analysis for location-based mobile services. In *International Conference on Software Engineering and Knowledge Engineering*, 2015.

[41] Tao Zhang, Jerry Gao, Jing Cheng, and Tadahiro Uehara. Compatibility testing service for mobile applications. In *Symposium on Service-Oriented System Engineering*, 2015.

[42] Mian Wan, Yuchen Jin, Ding Li, and William G. J. Halfond. Detecting display energy hotspots in android apps. In *International Conference on Software Testing, Verification and Validation*, 2015.

[43] Šarūnas Packevičius, Andrej Ušaniov, Šarūnas Stanskis, and Eduardas Bareiša. The testing method based on image analysis for automated detection of ui defects intended for mobile applications. In *International Conference on Information and Software Technologies*, 2015.

[44] Konstantin Knorr and David Aspinall. Security testing for android mhealth apps. In *Software Testing, Verification and Validation Workshops*, 2015.

[45] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *International Symposium on Software Testing and Analysis*, 2015.

[46] Guilherme de Cleva Farto and Andre Takeshi Endo. Evaluating the model-based testing approach in the context of mobile applications. In *Electronic Notes in Theoretical Computer Science*, 2015.

[47] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Scalable race detection for android applications. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[48] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. Mobiguitar: Automated model-based testing of mobile apps. In *IEEE Software*, 2015.

[49] Oum-EI-Kheir Aktouf, Tao Zhang, Jerry Gao, and Tadahiro Uehara. Testing location-based function services for mobile applications. In *Symposium on Service-Oriented System Engineering*, 2015.

[50] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *IEEE Symposium on Security and Privacy*, 2015.

[51] Behnaz Hassanshahi, Yaoqi Jia, Roland HC Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on android: Characterization and detection. In *European Symposium on Research in Computer Security*, 2015.

[52] Inês Coimbra Morgado and Ana CR Paiva. Testing approach for mobile applications through reverse engineering of ui patterns. In *International Conference on Automated Software Engineering Workshop*, 2015.

[53] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. Towards mutation analysis of android apps. In *International Conference on Software Testing, Verification and Validation Workshops*, 2015.

[54] Ana Rosario Espada, María del Mar Gallardo, Alberto Salmerón, and Pedro Merino. Runtime verification of expected energy consumption in smartphones. In *Model Checking Software*, 2015.

[55] Pingyu Zhang and Sebastian G. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. In *International Conference on Software Engineering*, 2014.

[56] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *International Conference on Software Testing, Verification, and Validation*, 2014.

[57] Chao-Chun Yeh, Han-Lin Lu, Chun-Yen Chen, Kee-Kiat Khor, and Shih-Kun Huang. Craxdroid: Automatic android system testing by selective symbolic execution. In *International Conference on Software Security and Reliability-Companion*, 2014.

[58] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. Intentfuzzer: detecting capability leaks of android applications. In *ACM symposium on Information, computer and communications security*, 2014.

[59] Sergiy Vilkomir and Brandi Amstutz. Using combinatorial approaches for testing mobile applications. In *International Conference on Software Testing, Verification, and Validation Workshops*, 2014.

[60] Hossain Shahriar, Sarah North, and Edward Mawangi. Testing of memory leak in android applications. In *International Symposium on High-Assurance Systems Engineering*, 2014.

[61] Sebastien Salva and Stassia R. Zafimiharisoa. Apset, an android application security testing tool for detecting intent-based vulnerabilities. In *International Journal on Software Tools for Technology Transfer*, 2014.

[62] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[63] Junfei Huang. Appacts: Mobile app automated compatibility testing service. In *International Conference on Mobile Cloud Computing, Services, and Engineering*, 2014.

[64] ChunHung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race detection for event-driven mobile applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[65] Chenkai Guo, Jing Xu, Hongji Yang, Ying Zeng, and Shuang Xing. An automated testing approach for inter-application security in android. In *International Workshop on Automation of Software Test*, 2014.

[66] Tobias Griebe and Volker Gruhn. A model-based approach to test automation for context-aware mobile applications. In *Annual ACM Symposium on Applied Computing*, 2014.

[67] Pedro Costa, Miguel Nabuco, and Ana C. R. Paiva. Pattern based gui testing for mobile applications. In *International Conference on the Quality of Information and Communications Technology*, 2014.

[68] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[69] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated analysis of adversarial android applications. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2014.

[70] Guillermo Suarez-Tangil, Mauro Conti, Juan E Tapiador, and Pedro Peris-Lopez. Detecting targeted smartphone malware with behavior-triggering stochastic models. In *European Symposium on Research in Computer Security*, 2014.

[71] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Working Conference on Mining Software Repositories*, 2014.

[72] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics*, 2014.

[73] Shuai Zhao, Xiaohong Li, Guangquan Xu, Lei Zhang, and Zhiyong Feng. Attack tree based android malware detection with hybrid analysis. In *International Conference on Trust, Security and Privacy in Computing and Communications*, 2014.

[74] Shengqian Yang, Dacong Yan, and Atanas Rountev. Testing for poor responsiveness in android applications. In *International Workshop on the Engineering of. Mobile-Enabled Systems*, 2013.

[75] Sarker T. Ahmed Rumee and Donggang Liu. Droidtest: Testing android applications for leakage of private information. In *International Journal of Information Security*, 2013.

[76] Vikrant Nandakumar, Vijay Ekambaram, and Vivek Sharma. Appstrument - a unified app instrumentation and automated playback framework for testing mobile applications. In *International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2013.

[77] Andrea Avancini and Mariano Ceccato. Security testing of the communication among android applications. In *International Workshop on Automation of Software Test*, 2013.

[78] Dacong Yan, Shengqian Yang, and Atanas Rountev. Systematic testing for resource leaks in android applications. In *International Symposium on Software Reliability Engineering*, 2013.

[79] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *International Workshop on Automation of Software Test*, 2012.

[80] Dominik Franke, Stefan Kowalewski, Carsten Weise, and Nath Prakobkosol. Testing conformance of life cycle dependent properties of mobile applications. In *International Conference on Software Testing, Verification and Validation*, 2012.

[81] Karthikeyan Balaji Dhanapal, K Sai Deepak, Saurabh Sharma, Sagar Prakash Joglekar, Aditya Narang, Aditya Vashistha, Paras Salunkhe, Harikrishna G. N. Rai, Arun Agrahara Somasundara,

and Sanjoy Paul. An innovative system for remote and automated testing of mobile phone applications. In *Service Research and Innovation Institute Global Conference*, 2012.

[82] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[83] Amiya K Maji, Fahad A Arshad, Saurabh Bagchi, and Jan S Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *International Conference on Dependable systems and Networks*, 2012.

[84] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *International Workshop on Automation of Software Test*, 2011.

[85] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 226–237. IEEE, 2016.

[86] Hammad Khalid, Meiyappan Nagappan, and Ahmed Hassan. Examining the relationship between findbugs warnings and end user ratings: A case study on 10,000 android apps. *IEEE Software*, 2015.

[87] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, 2013.

[88] Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, volume 3, page 260, 2003.

[89] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2013.

[90] Damien Octeau, Somesh Jha, Matthew Dering, Patrick Mcdaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43th Symposium on Principles of Programming Languages (POPL 2016)*, 2016.

[91] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *Proceedings of the 30th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC 2015)*, 2015.

[92] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*, 2014.

[93] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 37–48. IEEE, 2016.

[94] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *International Conference on Automated Software Engineering*, 2016.

[95] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *International Conference on Software Engineering*, 2016.

[96] Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Tan Hee Beng Kuan, and Charlie Soh. Achieving high code coverage in android ui testing via automated widget exercising. In *Asia-Pacific Software Engineering Conference*, 2016.

[97] Haowen Zhu, Xiaojun Ye, Xiaojun Zhang, and Ke Shen. A context-aware approach for dynamic gui testing of android applications. In *Computer Software and Applications Conference*, 2015.

[98] Kwangsik Song, Ah-Rim Han, Sehun Jeong, and Sung Deok Cha. Generating various contexts from permissions for testing android applications. In *International Conference on Software Engineering and Knowledge Engineering*, 2015.

[99] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. Sig-droid: Automated system input generation for android applications. In *International Symposium on Software Reliability Engineering*, 2015.

[100] Bo Jiang, Peng Chen, Wing Kwong Chan, and Xinchao Zhang. To what extent is stress testing of android tv applications automated in industrial environments? In *IEEE Transactions on Reliability*, 2015.

[101] Tobias Griebe, Marc Hesenius, and Volker Gruhn. Towards automated ui-tests for sensor-based mobile applications. In *International Conference on Intelligent Software Methodologies, Tools and Techniques*, 2015.

[102] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, and Porfirio Tramontana. Agrippin: a novel search based testing technique for android applications. In *International Workshop on Software Development Lifecycle for Mobile*, 2015.

[103] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *International Symposium on Software Testing and Analysis*, 2015.

[104] Inês Coimbra Morgado and Ana CR Paiva. The impact tool: Testing ui patterns on mobile applications. In *International Conference on Automated Software Engineering*, 2015.

[105] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[106] Ying-Dar Lin, Jose F. Rojas, Edward T.-H. Chu, and Yuan-Cheng Lai. On the accuracy, efficiency, and reusability of automated test oracles for android devices. In *IEEE Transactions on Software Engineering*, 2014.

[107] C.-J. Liang, N. Lane, N. Brouwers, L. Zhang, B. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao. Caiipa: Automated large-scale mobil app testing through contextual fuzzing. In *International conference on Mobile computing and networking*, 2014.

[108] Xiujiang Li, Yanyan Jiang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. User guided automation for testing mobile apps. In *Asia-Pacific Software Engineering Conference*, 2014.

[109] Clemens Holzmann and Patrick Hutflesz. Multivariate testing of native mobile applications. In *International Conference on Advances in Mobile Computing and Multimedia*, 2014.

[110] Xiangping Chen and Zhensheng Xu. Towards automatic consistency checking between web application and its mobile application. In *International Conference on Software Engineering and Knowledge Engineering*, 2014.

[111] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Ugo Gentile, Gianluca Mele, Roberto Nardone, and Valeria Vittorini. Improving code coverage in android apps testing by exploiting patterns and automatic test case generation. In *International workshop on Long-term industrial collaboration on software engineering*, 2014.

[112] Muhammad Adinata and Inggriani Liem. A/b test tools of native mobile application. In *International Conference on Data and Software Engineering*, 2014.

[113] Ying-Dar Lin, Edward T.-H. Chu, Shang-Che Yu, and Yuan-Cheng Lai. Improving the accuracy of automated gui testing for embedded systems. In *IEEE Software*, 2013.

[114] Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, 2013.

[115] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, 2013.

[116] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Nicola Amatucci. Considering context events in event-based testing of mobile applications. In *International Conference on Software Testing, Verification and Validation Workshops*, 2013.

[117] Antonio Corradi, Mario Fanelli, Luca Foschini, and Marcello Cinque. Context data distribution with quality guarantees for android-based mobile systems. In *Security and Communication Networks*, 2013.

[118] Sebastian Bauersfeld. Guidiff - a regression testing tool for graphical user interfaces. In *International Conference on Software Testing, Verification and Validation*, 2013.

[119] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis*, 2013.

[120] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. In *ACM SIGSOFT Software Engineering Notes*, 2012.

[121] Haeng-Kon Kim. Hybrid mobile testing model. In *International Conferences, ASEA and DRBC*, 2012.

[122] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2012.

[123] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based gui testing of an android application. In *IEEE International Conference on Software Testing, Verification and Validation*, 2011.

[124] Ben Sadeh, Kjetil Ørbekk, Magnus M. Eide, Njaal C.A. Gjerde, Trygve A. Tønnesland, and Sundar Gopalakrishnan. Towards unit testing of user interface code for android mobile applications. In *International Conference on Software Engineering and Computer Systems*, 2011.

[125] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A gui crawling-based technique for android mobile application testing. In *International Conference on Software Testing, Verification and Validation Workshops*, 2011.

[126] Zhifang Liu, Xiaopeng Gao, and Xiang Long. Adaptive random testing of mobile application. In *International Conference on Computer Engineering and Technology*, 2010.

[127] Milind G Limaye. *Software testing*. Tata McGraw-Hill Education, 2009.

[128] Software Testing Fundamentals. Software testing levels. http://softwaretestingfundamentals.com/software-testing-levels/.

[129] Afzal Wasif, Torkar Richard, and Feldt Robert. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51:957–976, 2009.

[130] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[131] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[132] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[133] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[134] Vincent F Taylor and Ivan Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 45–57. ACM, 2017.

[135] Mark Turner, Barbara Kitchenham, David Budgen, and OP Brereton. Lessons learnt undertaking a large-scale systematic literature review. In *Proceedings of EASE*, volume 8, 2008.

[136] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017.

[137] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.

[138] Samer Zein, Norsaremah Salleh, and John Grundy. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software*, 117:334–356, 2016.

[139] Abel Méndez-Porras, Christian Quesada-López, and Marcelo Jenkins. Automated testing of mobile applications: a systematic map and review. In *XVIII Ibero-American Conference on Software Engineering, Lima-Peru*, pages 195–208, 2015.

[140] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Bryan Robbins. Testing android mobile applications: Challenges, strategies, and approaches. *Advances in Computers*, 89(6):1–52, 2013.

[141] Jerry Gao, Wei-Tek Tsai, Rimi Paul, Xiaoying Bai, and Tadahiro Uehara. Mobile testing-as-a-service (mtaas)–infrastructures, issues, solutions and needs. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 158–167. IEEE, 2014.

[142] Oleksii Starov, Sergiy Vilkomir, Anatoliy Gorbenko, and Vyacheslav Kharchenko. Testing-as-a-service for mobile applications: state-of-the-art survey. In *Dependability Problems of Complex Information Systems*, pages 55–71. Springer, 2015.

[143] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. Software testing of mobile applications: Challenges and future research directions. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 29–35. IEEE, 2012.

[144] Marek Janicki, Mika Katara, and Tuula Pääkkönen. Obstacles and opportunities in deploying model-based gui testing of mobile software: a survey. *Software Testing, Verification and Reliability*, 22(5):313–341, 2012.

[145] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2017.

[146] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE Transactions on Software Engineering*, 2016.

[147] Ping Yan and Zheng Yan. A survey on dynamic mobile malware detection. *Software Quality Journal*, pages 1–29, 2017.

TABLE A1: The Full List of Examined Publications.

Year	Venue	Title
2016	APSEC	Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising
2016	ISSRE	Experience Report: Detecting Poor-Responsive UI in Android Applications
2016	ASE	Generating test cases to expose concurrency bugs in android applications
2016	AST	Fuzzy and cross-app replay for smartphone apps
2016	ICST	Automatically Discovering, Reporting and Reproducing Android Application Crashes
2016	JCST	Prioritizing Test Cases for Memory Leaks in Android Applications
2016	SecureComm	Using Provenance Patterns to Vet Sensitive Behaviors in Android Apps
2016	ICSE	Reducing combinatorics in GUI testing of android applications
2016	FSE	Minimizing GUI event traces
2016	ESORICS	Mobile Application Impersonation Detection Using Dynamic User Interface Extraction
2016	AST	Automated test generation for detection of leaks in Android applications
2016	ISSTA	Energy-aware test-suite minimization for android apps
2016	ISSREW	Replaying Harmful Data Races in Android Apps
2016	FSE	DiagDroid: Android performance diagnosis via anatomizing asynchronous executions
2016	ISSTA	Automatically verifying and reproducing event-based races in Android apps
2016	ICSE	Mobiplay: A remote execution based record-and-replay tool for mobile applications
2016	ISSTA	Sapienz: multi-objective automated testing for Android applications
2016	FSE	Automated test input generation for Android: are we really there yet in an industrial case?
2016	APSEC	Testing Android Apps via Guided Gesture Event Generation
2016	AST	Graph-aided directed testing of Android applications for checking runtime privacy behaviours
2016	ASE	Automated model-based android gui testing using multi-level gui comparison criteria
2016	ICSE	Mining Sandboxes
2016	MOBILESoft	Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring
2016	ISSTA	Monkey see, monkey do: effective generation of GUI tests with inferred macro events
2015	ISSTA	Systematic execution of Android test suites in adverse conditions
2015	ICST	Detecting Display Energy Hotspots in Android Apps
2015	OOPSLA	Scalable race detection for Android applications
2015	SEKE	Generating various contexts from permissions for testing Android applications
2015	ToR	To What Extent is Stress Testing of Android TV Applications Automated in Industrial Environments?
2015	SoMet	Towards Automated UI-Tests for Sensor-Based Mobile Applications
2015	ESORICS	Web-to-Application Injection Attacks on Android: Characterization and Detection
2015	ICIST	The Testing Method Based on Image Analysis for Automated Detection of UI Defects Intended for Mobile Applications
2015	MCS	Runtime Verification of Expected Energy Consumption in Smartphones
2015	ASEW	Testing Approach for Mobile Applications through Reverse Engineering of UI Patterns
2015	ICSTW	Towards mutation analysis of Android apps
2015	SOSE	Testing Location-Based Function Services for Mobile Applications
2015	IS	MobiGUITAR: Automated Model-Based Testing of Mobile Apps
2015	DeMobile	AGRippin: a novel search based testing technique for Android applications
2015	ICSTW	Security testing for Android mHealth apps
2015	SOSE	Compatibility Testing Service for Mobile Applications
2015	ISSRE	SIG-Droid: Automated System Input Generation for Android Applications
2015	COMPSAC	A Context-Aware Approach for Dynamic GUI Testing of Android Applications
2015	S&P	Effective Real-Time Android Application Auditing
2015	ISSTA	Dynamic detection of inter-application communication vulnerabilities in Android
2015	ASE	The iMPAcT Tool: Testing UI Patterns on Mobile Applications
2015	SEKE	Test Model and Coverage Analysis for Location-based Mobile Services
2015	ENTCS	Evaluating the Model-Based Testing Approach in the Context of Mobile Applications
2014	SPSM	A5: Automated Analysis of Adversarial Android Applications
2014	WISE	Improving code coverage in android apps testing by exploiting patterns and automatic test case generation
2014	MobiCom	Caiipa: Automated Large-scale Mobil App Testing through Contextual Fuzzing
2014	SAC	A model-based approach to test automation for context-aware mobile applications
2014	PLDI	Race detection for event-driven mobile applications
2014	AsiaCCS	IntentFuzzer: detecting capability leaks of android applications
2014	AST	An automated testing approach for inter-application security in Android
2014	ICSTW	Using Combinatorial Approaches for Testing Mobile Applications

2014	MoMM	Multivariate Testing of Native Mobile Applications
2014	STTT	APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities
2014	FSE	Detecting energy bugs and hotspots in mobile apps
2014	ICSE	Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain
2014	SEKE	Towards Automatic Consistency Checking between Web Application and its Mobile Application
2014	MobileCloud	AppACTS: Mobile App Automated Compatibility Testing Service
2014	ESORICS	Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models
2014	MSR	Mining energy-greedy API usage patterns in Android apps: an empirical study
2014	QUATIC	Pattern Based GUI Testing for Mobile Applications
2014	FSE	EvoDroid: segmented evolutionary testing of Android apps
2014	ICST	Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps
2014	TrustCom	Attack Tree Based Android Malware Detection with Hybrid Analysis
2014	SERE-C	CRAXDroid: Automatic Android System Testing by Selective Symbolic Execution
2014	HASE	Testing of Memory Leak in Android Applications
2014	WODA/PERTEA	Intent fuzzer: crafting intents of death
2014	PLDI	Race Detection for Android Applications
2014	APSEC	User Guided Automation for Testing Mobile Apps
2014	TSE	On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices
2014	ICODSE	A/B test tools of native mobile application
2013	FASE	A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications
2013	IJIS	DroidTest: Testing Android Applications for Leakage of Private Information
2013	ISSRE	Systematic testing for resource leaks in Android applications
2013	MOBIQUITOUS	Appstrument - A Unified App Instrumentation and Automated Playback Framework for Testing Mobile Applications
2013	IS	Improving the Accuracy of Automated GUI Testing for Embedded Systems
2013	OOPSLA	Targeted and depth-first exploration for systematic testing of android apps
2013	MOBS	Testing for poor responsiveness in android applications
2013	ESEC/FSE	Dynodroid: An Input Generation System for Android Apps
2013	ICST	GUIdiff - A Regression Testing Tool for Graphical User Interfaces
2013	AST	Security testing of the communication among Android applications
2013	ICSTW	Considering Context Events in Event-Based Testing of Mobile Applications
2013	SCN	Context data distribution with quality guarantees for Android-based mobile systems
2013	ICSE	Reran: Timing-and touch-sensitive record and replay for android
2013	OOPSLA	Guided GUI testing of android apps with minimal restart and approximate learning
2013	ISSTA	Automated testing with targeted event sequence generation
2012	SEN	Verifying android applications using Java PathFinder
2012	FSE	Automated concolic testing of smartphone apps
2012	ASEA/DRBC	Hybrid Mobile Testing Model
2012	AST	A whitebox approach for automated security testing of Android applications on the cloud
2012	SEN	Testing android apps through symbolic execution
2012	ICST	Testing Conformance of Life Cycle Dependent Properties of Mobile Applications
2012	SPSM	SmartDroid: an automatic system for revealing UI-based trigger conditions in android applications
2012	SRII	An Innovative System for Remote and Automated Testing of Mobile Phone Applications
2012	DSN	An empirical study of the robustness of Inter-component Communication in Android
2011	ICSTW	A GUI Crawling-Based Technique for Android Mobile Application Testing
2011	AST	Automating GUI testing for Android applications
2011	ICST	Experiences of System-Level Model-Based GUI Testing of an Android Application
2011	ICSECS	Towards Unit Testing of User Interface Code for Android Mobile Applications
2010	ICCET	Adaptive random testing of mobile application