Software Engineering for OpenHarmony: A Research Roadmap

LI LI, Beihang University, Beijing, China
XIANG GAO, School of Software, Beihang University, Beijing, China
HAILONG SUN, Beihang University, Beijing, China
CHUNMING HU, Beihang University, Beijing, China
XIAOYU SUN, Australian National University, Canberra, Australia
HAOYU WANG, Huazhong University of Science and Technology, Wuhan, China
HAIPENG CAI, Washington State University, Pullman, United States
TING SU, Software Institute, Shanghai Key Laboratory of Trustworthy Computing, shanghai, China
XIAPU LUO, Department of Computing, The Hong Kong Polytechnic University, Hong Kong, Hong Kong
TEGAWENDÉ BISSYANDE, SnT, University of Luxembourg, Luxembourg, Luxembourg
JACQUES KLEIN, SnT, University of Luxembourg, Luxembourg, Luxembourg
JOHN GRUNDY, Faculty of Information Technology, Monash University, Clayton, Australia
TAO XIE, Computer Science, Peking University, Beijing, China
HAIBO CHEN, Shanghai Jiao Tong University, Shanghai, China
HUAIMIN WANG, Natl Univ Def Technol, Changsha, China

Mobile software engineering has been a hot research topic for decades. Our fellow researchers have proposed various approaches (with over 7,000 publications for Android alone) in this field that essentially contributed to the great success of the current mobile ecosystem. Existing research efforts mainly focus on popular mobile platforms, namely Android and iOS. OpenHarmony, a newly open-sourced mobile platform, has rarely been considered, although it is the one requiring the most attention as OpenHarmony is expected to occupy one-third of the market in China (if not in the world). To fill the gap, we present to the mobile software engineering community a research roadmap for encouraging our fellow researchers to contribute promising approaches to OpenHarmony. Specifically, we start by presenting a tertiary study of mobile software engineering, attempting to understand what problems have been targeted by the mobile community and how they have been

Authors' Contact Information: Li Li, Beihang University, Beijing, Beijing, China; e-mail: lilicoding@ieee.org; Xiang Gao, School of Software, Beihang University, Beijing, Beijing, China; e-mail: xiang_gao@buaa.edu.cn; Hailong Sun, Beihang University, Beijing, Beijing, China; e-mail: sunhl@buaa.edu.cn; Chunming Hu, Beihang University, Beijing, Beijing, China; e-mail: hucm@buaa.edu.cn; Xiaoyu Sun, Australian National University, Canberra, Australian Capital Territory, Australia; e-mail: xiaoyu.sun1@anu.edu.au; Haoyu Wang, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: haoyuwang@hust.edu.cn; Haipeng Cai, Washington State University, Pullman, Washington, United States; e-mail: haipengc@buffalo.edu; Ting Su, Software Institute, Shanghai Key Laboratory of Trustworthy Computing, shanghai, China; e-mail: tsuletgo@gmail.com; Xiapu Luo, Department of Computing, The Hong Kong Polytechnic University, Hong Kong, Hong Kong; e-mail: csxluo@comp.polyu.edu.hk; Tegawendé Bissyande, SnT, University of Luxembourg, Luxembourg, Luxembourg; e-mail: tegawende.bissyande@uni.lu; Jacques Klein, SnT, University of Luxembourg, Luxembourg, Luxembourg; e-mail: jacques.klein@uni.lu; John Grundy, Faculty of Information Technology, Monash University, Clayton, Victoria, Australia; e-mail: john.grundy@monash.edu; Tao Xie, Computer Science, Peking University, Beijing, China; e-mail: taoxie@pku.edu.cn; Haibo Chen, Shanghai Jiao Tong University, Shanghai, China; e-mail: haibochen@sjtu.edu.cn; Huaimin Wang, Natl Univ Def Technol, Changsha, China; e-mail: whm_w@163.com.

ACM 1557-7341/2025/2-ART
https://doi.org/10.1145/3720538

resolved. We then summarize the existing (limited) achievements of OpenHarmony and subsequently highlight the research gap between Android/iOS and OpenHarmony. This research gap eventually helps in forming the roadmap for conducting software engineering research for OpenHarmony.

1 Introduction

Mobile Software Engineering has been a hot topic for many years. It concerns all the aspects of software engineering in mobile, including the design, development, validation, execution, and evolution of mobile applications. This has been considered extremely important as nowadays our lives have been empowered by the massive increase in the use of mobile apps. Indeed, the number of mobile devices will reach 7 billion in 2023. The number of mobile apps that can be run on each mobile device (for both Android and iOS) has exceeded the 2 million mark. Furthermore, these figures are constantly increasing, thanks to app stores and marketplaces that allow users to effortlessly download and install applications.

Mobile platforms are rapidly evolving as well in order to continuously integrate diverse and powerful capabilities, including various sensors, cameras, wireless communication channels, as well as on-device memory and disk capacities. As a result of ingeniously applying these technological developments, developers of mobile software are pushing the boundaries with innovative mobile services and exciting mobile applications. Consequently, due to the rapid development and evolution of mobile software, developers face new software engineering challenges.

To address these challenges, researchers in the software engineering community have explored various research directions and developed lots of novel tools supported by formally grounded methods. Indeed, researchers have proposed various static program analysis approaches (i.e., by just scanning the code without actually running mobile apps) for characterizing issues (including ones related to mobile security, compatibility, energy consumption, etc.) of mobile apps [56]. For example, Arzt et al. [8] have designed and developed the famous FlowDroid approach that performs static taint analysis of Android apps for pinpointing privacy leaks. Except static analysis approaches, researchers have also invented various dynamic testing approaches (i.e., by actually running mobile apps on devices) for detecting potential defects of mobile apps at runtime [51]. For example, Amalfitano et al. [6] have proposed a GUI ripping approach for automated testing of Android apps. Su et al. [102] have proposed to achieve the same purpose through a model-based approach. The aforementioned research approaches have contributed to the huge success of the current flourishing mobile ecosystem, including both Android and iOS.

Unfortunately, these approaches cannot directly benefit OpenHarmony[1], which is a new open-sourced mobile platform launched by the OpenAtom Foundation after receiving a donation of the open-source code from Huawei.[2] These approaches, theoretically, should be generic and hence should also work for OpenHarmony. However, significant engineering efforts are still required to achieve that due to the following reasons (more details will be given in the background section): (1) The Openharmony platform empowers a new framework supported by a layered architecture; (2) Openharmony apps are written in a newly designed language called ArkTS.

Unlike Android and iOS, which have been well-established for many years and each has a thriving ecosystem to support their growth, the development of the Openharmony ecosystem is still at an earlier stage. We, therefore, argue that OpenHarmony requires more help from the software engineering research community.[3] We call on actions for conducting software engineering research for OpenHarmony.

As our initial attempt, we decided to present to the community an initial research roadmap for guiding our mobile software engineering community in achieving that. Specifically, we start by conducting a tertiary study to understand the current achievements (i.e., Section 2) achieved by the Mobile Software Engineering community. We then discuss the current state of the OpenHarmony ecosystem, followed by a comparative study to locate the

[1]https://www.openharmony.cn

[2]More background info: https://lilicoding.github.io/resources/Background_of_OpenHarmony.pdf

[3]This could be regarded as new opportunities for the mobile software engineering community.

technical gaps between mature platforms and OpenHarmony (i.e., Section 3). Based on that, we summarize the technical deficiencies of OpenHarmony and propose a roadmap for our research community to complete (i.e., Section 4). After that, we discuss the possible challenges and future research opportunities faced by conducting software engineering research for OpenHarmony (i.e., Section 5), before discussing the related work in Section 6 concluding this paper in Section 7.

2 Tertiary Study on Mobile Software Engineering

In this work, we are interested in building a research roadmap for conducting software engineering research for OpenHarmony. Unfortunately, since OpenHarmony is still in its early stages, there is not much work proposed for that. As what happened for its counterparts (Android or iOS), there will be huge software engineering issues that need to be addressed before establishing a mature ecosystem. We hence resort to learning from the Android ecosystem to form the research roadmap to guide software engineering studies for OpenHarmony. The rationale behind this decision is that we believe all the research efforts contributed to improving the Android and iOS ecosystem could be also conducted for OpenHarmony.

In this work, we resort to a tertiary study to understand the status quo of mobile software engineering research. Tertiary studies, which summarize and synthesize findings from existing systematic reviews and meta-analyses on a specific topic (cf. [53]), align with the main objective of this project: to inform and guide the research direction of Open Harmony by examining existing work in the field of mobile SE. Here, we remind readers that the choice of conducting a tertiary study rather than other types of studies because it allows for a comprehensive synthesis of existing research, providing a broader perspective on the challenges and trends for research in OpenHarmony. This approach aligns with our research objectives to identify gaps and future research directions in this field.

2.1 Tertiary Study Method

In this work, we conduct a tertiary study following the methodologies outlined by Kitchenham and Charters et al. [46]. This approach employs the same methods as a typical systematic literature review (SLR) but focuses on collecting secondary studies. Specifically, the working process consists of three main phases: *planning*, *conducting*, and *reporting*. In the planning phase, we determined the key elements of the tertiary study protocol (including research questions (RQs), search keywords, selected databases, quality assessment, and studies selection criteria), which was reviewed and agreed upon by all authors. In all manual activities requiring human judgment, we followed the data extraction and checking approach suggested by Brereton et al. [17], with the second author acting as the extractor and the first author as the checker. The entire review method is depicted in Fig. 1, adhering to the guidelines for systematic studies to visualize the adopted review process [106].

2.1.1 ***Planning.*** The planning phase aims to complete the first two steps highlighted in Fig. 1, namely RQ Identification and Keywords Identification.

[RQ Identification] The goal is to analyze secondary studies of mobile software engineering for the purpose of identifying the best research advancements with respect to target problems and solutions. We thus defined the following RQ on top of this objective:

RQ: What problems are targeted by our fellow researchers in the MSE community and how they are resolved?

[keywords Identification] Then, in the next step, we identify the search keywords that could be used to find all the relevant publications, in order to answer the pre-defined research questions. In line with the approach of other tertiary studies [11, 49], we resort to considering the existing survey and literature review papers (i.e., secondary studies), for which our fellow researchers have already systematically reviewed the different aspects of mobile software engineering. We believe these survey papers are representative of the status quo of mobile software engineering research. To this end, we identify the search keywords based on these concerns. Table 1

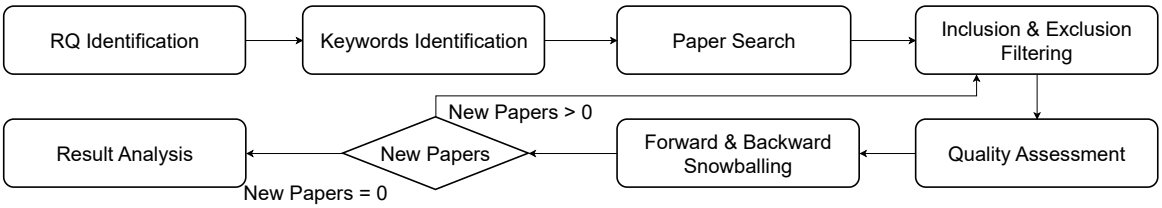

Fig. 1. The working process of our systematic literature review.

depicts the list of identified keywords. In total, we have identified two groups of keywords: keywords related to mobile, and secondary studies (i.e., G1 and G2), respectively. Regarding the mobile field, we include the keywords on top of its definition, which most commonly refers to smartphones such as Android, IOS and Phone. In addition, to assemble the keywords for secondary studies, we carried a group of 35 keywords forward in this work from a tertiary study on systematic literature reviews (SLRs) in software engineering by kotti et al. [53]. We then form the query based on this rule [4] for which we require it to contain at least one keyword from each group.

Table 1. Repository Search Keywords.

Group (and)	Keywords (or)
G1	Mobile, Android, iOS, *phone*
G2	analysis of research; body of published research; centralized tutorial; common practices; comparative study; conceptual analysis; editorial; editor's preview; evidence-based software engineering; in-depth analysis; literature analysis; literature review; literature survey; lookup table; manifesto; meta-analysis; meta-survey; methodologies; past studies; review of studies; strategic directions; structured review; study; subject matter expert; survey and classification; survey; systematic approach; systematic mapping study; systematic review; taxonomy

2.1.2 ***Conducting.*** The conducting phase aims to complete the following four steps highlighted in Fig. 1: Paper Search, Inclusion & Exclusion Filtering, Quality Assessment, and Forward & Backward Snowballing. We now detail these steps, respectively.

[Paper Search] After the query is formed, in **Step 3**, we directly applied to search relevant studies in the following four online digital libraries for systematic querying: IEEE Xplore, ACM Digital Library, Science Direct, and Springer, and leveraged another two indexing databases (i.e., DBLP and Scopus) for cross-comparison. These databases were selected based on their comprehensive coverage of scholarly publications in the field of computer science.

Unfortunately, many of the located papers were either of low quality or outside our area of focus. To narrow down to the most relevant studies, in **Step 4**, we refine the gathered list of relevant papers manually to ensure their relevance to mobile software engineering (i.e., could indeed be helpful for answering the aforementioned research question). Specifically, we evaluated all the collected papers using the following set of inclusion and exclusion criteria (IC/EC) after reviewing their titles, keywords, and abstracts.

[Inclusion and Exclusion Filtering] The following set of IC/EC was applied to all papers obtained through the search strategy to ensure that only relevant secondary studies were included in this tertiary study.

Inclusion Criteria.

(1) Only secondary studies are included. In other words, this encompasses research such as systematic literature reviews (SLRs), systematic mapping studies, and meta-analyses that follow documented systematic methods.

[4] ($g1_1$ OR ... OR $g1_x$) AND ($g2_1$ OR ... OR $g2_y$), where $g1_i \in G1$, $g2_j \in G2$ and $1 \leq i \leq x$, $1 \leq j \leq y$, for which x and y are the number of keywords in G1 and G2, respectively

(2) Publications must address fundamental software engineering topics, including but not limited to software development methodologies, tools, and practices [120].
(3) To ensure the relevance of the review, we focused on publications from the last 2013 years, depending on the field's development and trends.

Exclusion Criteria.

(1) Since we only include survey or literature review papers, all the non-survey papers are simply excluded from our study.
(2) Papers for which the PDFs cannot be found are excluded.
(3) Publications not written in English are also excluded.
(4) Although there are some papers that meet our selection criteria (i.e., whose title contains the group keywords in Table 1), their topics may not strictly fall into the software engineering category. [5] To this end, publications that do not fall within the software engineering category are excluded.
(5) Short papers (i.e., less than eight pages in double-column format or 11 pages in single-column format) were excluded.

After the IC/EC is formed, we manually apply these IC/EC to filter out irrelevant instances. In line with the adopted guidelines [46], the selection process was based on the titles, author keywords, and abstracts of the papers. We start by identifying two participants, namely the data extractor and the data checker. These two participants will first independently review a set of 30 randomly selected studies to determine their consensus on the inclusion or exclusion criteria. Their level of agreement is measured using Cohen's Kappa statistic [86], assessed inter-rater reliability. Any discrepancies were resolved to reach a consensus. This process was repeated until a Kappa score of at least 0.8 was achieved. This ensured that both participants shared the same understanding of the IC/EC, allowing them to fairly review the remaining large number of studies. As a result, a total of 143 distinct secondary studies were retained.

Table 2. DARE-4 Criteria for Quality Assessment.

QA Criterion	Assessment	Score	Description
Inclusion & Exclusion	Yes	1	Explicit definition of IC/EC
	Partial	0.5	Implicit definition of IC/EC
	No	0	No IC/EC defined
Search space	Yes	1	4+ digital libraries searched and snowballing search strategies applied
	Partial	0.5	3-4 digital libraries searched and snowballing search strategies applied
	No	0	1-2 digital libraries searched
Quality assessment of primary studies	Yes	1	Quality criteria explicitly described and applied
	Partial	0.5	Implicit quality assessment
	No	0	No quality assessment
Information regarding primary studies	Yes	1	Complete information presented about primary studies
	Partial	0.5	Summary information presented about primary studies
	No	0	Results of primary studies not specified

[Quality Assessment] We then conducted a manual quality assessment of the 143 selected secondary papers to ensure the reliability of our study results. In this step, we adhered to a recommended quality assessment process for tertiary studies [50], utilizing the DARE-4 criteria as outlined in Table 2, following the most recent tertiary study and relevant to the goals of this work.

[5]For example, the paper entitled "A Taxonomy and Survey of Microscopic Mobility Models from the Mobile Networking Domain" is excluded because its primary focus is on mobile network simulations and the development of realistic mobility models. Although it includes the survey and mobile keywords, it is not really in the domain of software engineering and hence is excluded.

Specifically, the DARE-4 criteria are based on four key questions, each of the questions is scored as Y (yes-1 point), P (partially-0.5 point), or N (no-0 points). The total score for a study is the sum of these points, with a maximum possible score of four and a minimum of zero. Studies must score at least two points to be included.

In addition, we adhered to a systematic data extraction and checking process, achieving an inter-rater agreement of 82%. Most disagreements occurred on the last question, which involves the information provided about the reviewed primary studies due to the subjective nature of this question. As a result, 26 out of 143 studies (18.18%) were excluded for scoring less than two. The total scores for accepted studies are shown in Table 3. We noted that the excluded lower-quality secondary studies often lacked clear documentation of inclusion/exclusion criteria, did not specify search sources, or failed to assess the quality of the included primary studies.

[Backward and Forward Snowballing] After filtering out irrelevant papers, we conduct forward and backward snowballing (i.e., **Step 5**) by reviewing all referenced papers to determine if they should be included in our study. Both backward and forward snowballing were applied. The backward snowballing involved reviewing the references of the included papers, while forward snowballing examined papers that cited the included studies. This approach ensured comprehensive coverage of the relevant literature. Two iterations of snowballing were conducted. In each iteration, the papers identified through snowballing were subjected to the same inclusion and exclusion criteria, ensuring consistency in the selection process. Additionally, we have cross-checked the results (i.e., **Step 6**) from the previous two steps (i.e., inclusion and exclusion criteria filtering and quality assessment) to ensure the reliability of our findings.

2.1.3 ***Reporting.*** To form the final report, we extract the following information from each of the quality-accepted secondary studies.

- *Title and Source*: The publication's title and its source, including journal, workshop proceedings, conference proceedings, or book chapter.
- *Publication Year*: To track the annual evolution and research interest in ML4SE.
- *Publication Venue*: To identify key publishers within this specific area of research.
- *Author Names, Institutions, and Countries*: To recognize leading research teams and their geographical distribution.
- *Target Problem*: To examine the problem targeted by secondary studies.
- *Research Method*: To examine the techniques most commonly adopted by secondary studies to solve the target problems.

[Result Analysis] We were able to eventually collect 51 papers to answer our research question defined at the beginning of this study. Table 3 enumerates the list of selected papers, including their publication year and venue. Once the relevant papers are collected, we carefully read all of them and attempt to extract the relevant data (i.e., **Step 7**) from each paper to answer the research question. Specifically, we aim to extract the following two types of information: (1) Targeted Problems, which involve understanding the issues within the Android/iOS ecosystem that have been identified by our MSE researchers as problems needing resolution to create a more user-friendly mobile ecosystem, and (2) Fundamental Techniques, aimed at discovering the techniques required to address the various challenges in the mobile community. Considering that OpenHarmony may encounter similar issues to those faced by Android and iOS, we argue that insights gained from exploring these two aspects could prove valuable in shaping the roadmap for conducting software engineering research for OpenHarmony. Furthermore, similar to our approach in identifying relevant papers, we have conducted cross-checks of our observations, involving at least two authors, to ensure the reliability of these observations, thereby enhancing the trustworthiness of the research roadmap.

Table 3. The List of Selected Publications.

Authors	Title	Year	Venue
Senanayake et al. [96]	Android Source Code Vulnerability Detection: A Systematic Literature Review	2023	CSUR
Wu et al. [112]	A systematic literature review on Android-specific smells	2023	JSS
Liu et al. [67]	Deep Learning for Android Malware Defenses: A Systematic Literature Review	2022	CSUR
Júnior et al. [44]	Dynamic Testing Techniques of Non-Functional Requirements in Mobile Apps: A Systematic Mapping Study	2022	CSUR
Delgado-Santos et al. [26]	A Survey of Privacy Vulnerabilities of Mobile Device Sensors	2022	CSUR
Lee et al. [55]	A Systematic Survey on Android API Usage for Data-Driven Analytics with Smartphones	2022	CSUR
Nakamura et al. [75]	What factors affect the UX in mobile apps? A systematic mapping study on the analysis of app store reviews	2022	JSS
Wimalasooriya et al. [111]	A systematic mapping study addressing the reliability of mobile applications: The need to move beyond testing reliability	2022	JSS
Zhan et al. [123]	Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review	2021	TSE
Shamsujjoha et al. [98]	Developing Mobile Applications Via Model Driven Development: A Systematic Literature Review	2021	IST
Ebrahimi et al. [28]	Mobile app privacy in software engineering research: A systematic mapping study	2021	IST
De Munk and Malavolta [25]	Measurement-based Experiments on the Mobile Web: A Systematic Mapping Study	2021	EASE
Yasuda et al. [119]	Autonomous Visual Navigation for Mobile Robots: A Systematic Literature Review	2020	CSUR
Luo et al. [69]	A Survey of Context Simulation for Testing Mobile Context-Aware Applications	2020	CSUR
C. et al. [19]	Energy Diagnosis of Android Applications: A Thematic Taxonomy and Survey	2020	CSUR
Qiu et al. [89]	A Survey of Android Malware Detection with Deep Neural Models	2020	CSUR
Li et al. [58]	Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark	2019	TSE
Al-Subaihin et al. [2]	App store effects on software engineering practices	2019	TSE
Kaur and Kaur [45]	Investigation on test effort estimation of mobile applications: Systematic literature review and survey	2019	IST
Barmpatsalou et al. [12]	Current and Future Trends in Mobile Device Forensics: A Survey	2018	CSUR
Biørn-Hansen et al. [16]	A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development	2018	CSUR
Jabangwe et al. [42]	Software engineering process models for mobile app development: A systematic literature review	2018	JSS
Ahmad et al. [1]	Perspectives on usability guidelines for smartphone applications: An empirical investigation and systematic literature review	2018	IST
Kim et al. [48]	A Survey on Recent OS-Level Energy Management Techniques for Mobile Processing Units	2018	TPDS
Kong et al. [52]	Automated Testing of Android Apps: A Systematic Literature Review	2018	TRel
Genc-Nayebi and Abran [37]	A systematic literature review: Opinion mining studies from mobile app store user reviews	2017	JSS
Li et al. [56]	Static analysis of android apps: A systematic literature review	2017	IST
Xu et al. [113]	Toward Engineering a Secure Android Ecosystem: A Survey of Existing Techniques	2016	CSUR
Martin et al. [73]	A survey of app store analysis for software engineering	2016	TSE
Zein et al. [121]	A systematic mapping study of mobile application testing techniques	2016	JSS
Sufatrio et al. [103]	Securing Android: A Survey, Taxonomy, and Challenges	2015	CSUR
Hoseini-Tabatabaei et al. [41]	A survey on smartphone-based systems for opportunistic user context recognition	2013	CSUR
Pereira and Rodrigues [85]	Survey and analysis of current mobile learning applications and technologies	2013	CSUR
Shahzad et al. [97]	Socio-technical challenges and mitigation guidelines in developing mobile healthcare applications	2017	JMIHI
Ali et al. [4]	Self-adaptation in smartphone applications: Current state-of-the-art techniques, challenges, and future directions	2021	DKE
Autili and others. [10]	Software engineering techniques for statically analyzing mobile apps: research trends, characteristics, and potential for industrial adoption	2021	JISA
Silva et al. [100]	A mapping study on mutation testing for mobile applications	2022	STVR
Hort et al. [40]	A survey of performance optimization for mobile applications	2021	TSE
Maniriho et al. [71]	A Survey of Recent Advances in Deep Learning Models for Detecting Malware in Desktop and Mobile Platforms	2024	CSUR
Qiu et al. [88]	Differentiated Location Privacy Protection in Mobile Communication Services: A Survey from the Semantic Perception Perspective	2023	CSUR
Silva et al. [99]	A survey on the tool support for the automatic evaluation of mobile accessibility	2018	MODELSWARD
Yan and Yan [118]	A survey on dynamic mobile malware detection	2018	SQJ
Altaleb and Gravell [5]	Effort Estimation across Mobile App Platforms using Agile Processes: A Systematic Literature Review	2018	JoS
Wang et al. [109]	Runtime Permission Issues in Android Apps: Taxonomy, Practices, and Ways Forward	2022	TSE
Sadeghi et al. [93]	A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software	2016	TSE
Nie et al. [76]	A systematic mapping study for graphical user interface testing on mobile apps	2023	IET
Tramontana et al. [105]	Automated functional testing of mobile applications: a systematic mapping study	2019	SQJ
Zein et al. [122]	Systematic reviews in mobile app software engineering: A tertiary study	2023	IST
Zhan et al. [124]	A Comparative Study of Android Repackaged Apps Detection Techniques	2019	SANER
Sadeghi et al. [93]	A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software	2016	TSE
Tramontana et al. [105]	Automated functional testing of mobile applications: a systematic mapping study	2019	SQJ

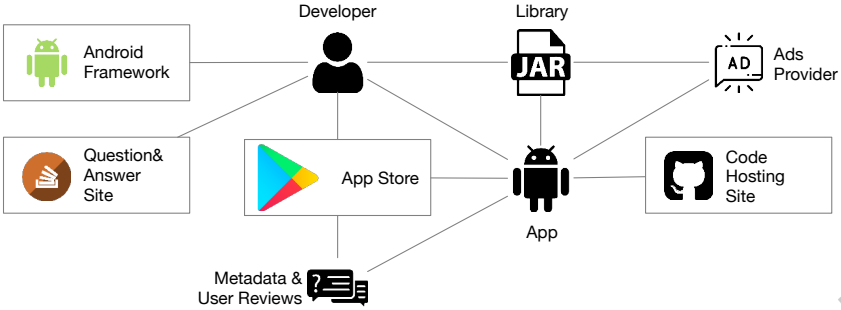

Fig. 2. Overview of the Major Participants (or Artifacts) Involved in MSE Research.

2.2 Problem

Before going into the details in summarizing the top problems targeted by our fellow researchers in MSE, we first present the major participants (or artifacts) involved in MSE research. These participants have been closely associated with the top problems identified and handled in MSE. As illustrated in Fig. 2, **developers** play a core role in MSE and contribute to the ecosystem by implementing **mobile apps** based on the **Android framework** (also known as the SDK) provided by Google, along with various **third-party libraries** that are pre-developed for facilitating app developments. The libraries also include the ones used to provide **advertisements**, which also play a crucial role in Android as they are the major source for app developers to make profits.[6] When there are problems encountered while developing an app, developers frequently resort to **question and answer website** (such as Stack Overflow) to search for solutions. The app's source code is often managed on code hosting websites such as Github, which is also one of the most important resources leveraged by mining software repository researchers to learn for improving Android apps. Once the apps are developed, they will be uploaded to **app stores** such as the official Google Play store, on which various **metadata** associated with the app (such as app's description, name, authors, etc.) will also be provided. The app stores are the main portal for users to find and install apps. Except for searching and installing apps, app stores also provide a platform for users to leave feedback (i.e., **user comments**, which could be complaints about defects or suggestions regarding new app features) for their apps on dedicated pages.

We now highlight the top problems targeted by our fellow researchers (cf. Table 4). These top problems could be applied to any of the aforementioned participants highlighted in Fig. 2. The problems are mainly grouped into nine categories, including app development, app deployment, user experience, security and privacy, quality, reliability, performance, energy, and socio-technical issues. To help readers better understand each of the categories (i.e., the actual problems handled by our fellow researchers), we also provide various problem examples in the second column of the table.

2.3 Technique

To solve the above software engineering problems, researchers have proposed various kinds of techniques. Note that, while there are more techniques designed to solve the above problems, e.g., trust environment execution (TEE) for increasing mobile application security, we will not include them but only consider the software engineering techniques in this work. Also, resolving software engineering tasks often involves manual efforts, such as confirming the warnings yielded by static analyzers or labelling datasets for training machine learning

[6]Indeed, app developers often cannot make profits directly from the apps per se as they are often made available to users as free apps.

Table 4. The top problems targeted by the examined papers.

Category	Problem Examples	Papers
App Development	Representative problems include (1) Learning new requirements by analyzing user comments, (2) Facilitating app developments by recommending third-party libraries, APIs, and code snippets, (3) Generating code for GUI components, (4) Facilitating app testing by automatically generating test cases, etc.	[2, 4, 5, 10, 16, 19, 28, 37, 40–42, 44, 45, 48, 52, 56, 67, 71, 75, 76, 85, 89, 93, 93, 96, 98–100, 103, 105, 105, 109, 113, 118, 121, 122, 124]
App Deployment	Problems related to app deployment include (1) Supporting code obfuscation, (2) Supporting app hardening, and (3) Supporting obfuscation for AI models inside apps.	[2, 4, 41, 42, 52, 85, 103, 113]
User Experience	Example problems include (1) Optimizing user experience by analyzing end-user perception, (2) Understanding user satisfaction by analyzing user reviews (feedback on app stores), and (3) Characterizing human-centric issues related to the success of apps.	[1, 2, 4, 5, 16, 19, 25, 26, 28, 37, 41, 42, 44, 48, 52, 55, 69, 75, 76, 85, 103, 105, 109, 109]
Security and Privacy	Representative problems include (1) Detecting privacy leaks, (2) Discovering sensitive hidden behaviors, (3) Exploiting component hijack attacks, (4) Exploring privilege escalation attacks, (5) Uncovering cryptographic API misuses, (6) Predicting malware and its families, etc.	[4, 10, 12, 16, 26, 28, 40, 44, 52, 55, 56, 58, 67, 71, 88, 89, 93, 93, 100, 103, 109, 109, 113, 118, 121, 123, 124]
Quality	Representative problems include (1) Detecting and fixing concurrency errors in mobile apps, (2) Characterizing the app's maintainability by understanding the evolution of deprecated APIs, the usage of incompatible APIs, (3) Improving effectiveness and efficiency of app testing approaches by automatically generating better test cases, estimating test efforts and prioritizing test cases.	[2, 4, 5, 10, 12, 16, 19, 25, 37, 40, 41, 44, 45, 52, 56, 58, 69, 75, 76, 85, 89, 93, 93, 96, 98, 105, 105, 111–113, 118, 122–124]
Reliability	Targeted problems include understanding, locating, and automatically repairing app crashes (caused by API misuses, and compatibility issues), failures, exceptions, and runtime errors.	[1, 10, 12, 16, 19, 26, 40, 44, 45, 52, 67, 69, 71, 76, 85, 89, 93, 96, 98, 100, 105, 105, 109, 111–113, 119, 122, 124]
Performance	Performance-related problems include (1) Assuring the app's efficiency by detecting and refactoring code smells and (2) Summarizing performance anti-patterns and their potential improving counterparts.	[2, 4, 5, 10, 16, 19, 25, 40, 41, 44, 45, 48, 52, 67, 69, 85, 96, 98, 100, 103, 112, 113, 118, 119, 124]
Energy	Energy Management problems include (1) Adjusting power states of processing units (2) Exploiting computing resources, and (3) Characterizing and detecting energy issues (e.g., bugs, leaks, hogs, hotspots, wakelock, sensors, network, and display).	[4, 10, 19, 40, 44, 48, 52, 85, 100, 103, 109]
Socio-technical issues	Targeted problems include (1) Understanding why mobile app users do not adopt security precautions in the smartphone context and studying how to use media campaigns to raise user awareness of security issues and (2) Identifying the common risks that hinder mobile application development in the healthcare domain and the mitigating strategies against those risks.	[4, 12, 45, 75, 97, 99, 124]

models, etc. In this work, we will not take into account those manual approaches. For the remaining techniques, after discussing them among co-authors, we preliminarily categorize them as static-based, dynamic-based, and learning-based approaches. Fig. 3 highlights the represented ones.

Static Approaches. Static approaches are the analysis of programs performed without executing them. The widely used static approaches are listed in Fig. 3. These static approaches have been applied to the SE problems of mobile applications, Android frameworks and mobile operating systems. Specifically, static approaches (e.g., taint analysis, symbolic execution, code instrumentation, model checking) are widely used to detect application bugs, including functional errors, code smells, security weaknesses/vulnerabilities, energy and performance bugs, permission escalations, etc. Beyond bug detection, static approaches (e.g., application hardening, code sign) are also used to increase the security and reliability of mobile applications. Moreover, with the rapid development of machine/deep learning, we have observed a trend to use static approaches to extract program features, which are then provided to learning approaches.

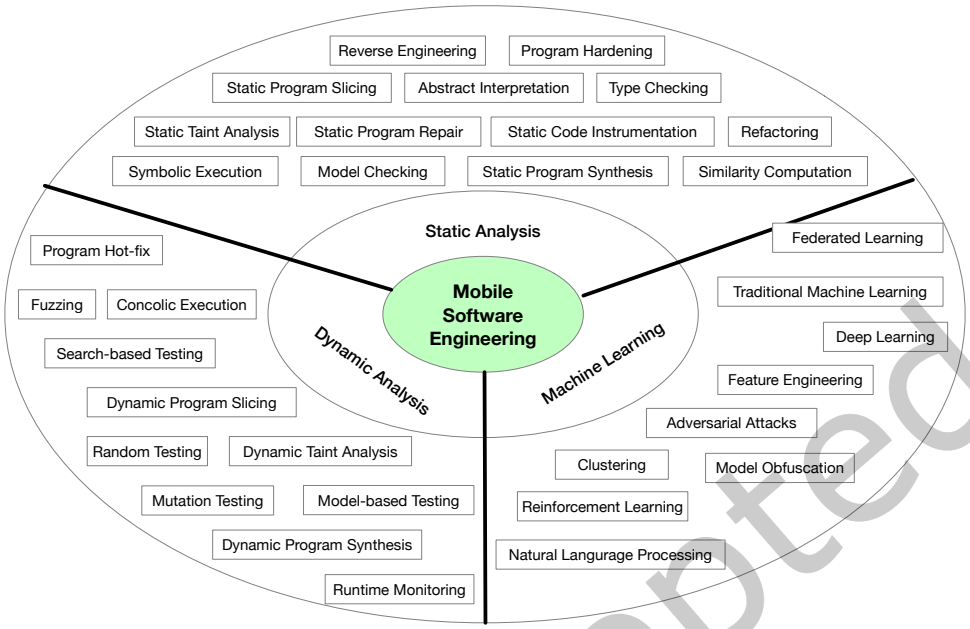

Fig. 3. Overview of the Representative Techniques Adopted in MSE.

Dynamic approaches. In contrast with static approaches, dynamic approaches are performed on programs during their execution. Similar to static approaches, dynamic approaches are also applied for program testing. Widely used dynamic testing techniques include search-based testing, black-box/random testing, grey-box fuzzing, concolic execution, event-driven test generation, mutation testing, etc. Dynamic program analysis is also applied for security analysis (e.g., dynamic taint analysis and runtime monitoring) and automated program repair.

Learning-based approaches Beyond the traditional static and dynamic approaches, we have seen an increasing trend that applies machine/deep learning techniques to solve mobile software engineering problems. Learning techniques train models by extracting features from large program artifacts and have achieved significant success in the field of code analysis. Learning-based techniques have been applied to solve many mobile software engineering tasks, including vulnerability detection, privacy issues detection, program testing, code smell checking etc. Moreover, it has recently garnered considerable research attention to employ deep learning techniques to thwart Android malware attacks.

3 The State of the OpenHarmony Ecosystem

As revealed in the previous literature review, despite Mobile Software Engineering being a longstanding and hot topic, the efforts spent by our fellow researchers for exploring OpenHarmony have been limited. Indeed, there is almost no contribution made to OpenHarmony in the current MSE community. Therefore, as our initial attempt towards bringing OpenHarmony research to the Mobile Software Engineering (MSE) community, we summarize the current achievements of OpenHarmony to help readers better understand the state of the OpenHarmony ecosystem. Specifically, in this section, we briefly introduce the existing toolchains and datasets available in the community. We then go one step further to summarize other existing resources that may not be directly related to OpenHarmony but could still be beneficial to grow the ecosystem of OpenHarmony.

3.1 Existing Toolchains

We then look at the existing toolchains offered by the official OpenHarmony framework to support app developments and these toolchains are considered important and essential. Indeed, these tools could provide fundamental capabilities to support the implementation of more advanced OpenHarmony-specific toolchains. Ideally, these toolchains should cover the full lifecycle of app development, including development, build, testing, debugging, code review, and publishing. Table 5 summarizes some of the tools provided by OpenHarmony. The second column demonstrates the software engineering phase that the tool intends to support. At the moment, these toolchains have covered almost all the aforementioned lifecycle phases, e.g., including app development-related ones (e.g., IDE, Emulator, Device Manager), app building tools (e.g., hvigor), app testing tools (e.g., jsunit, uitest), debugging tools (e.g., HiLog, profiler), code reviewing tools (e.g., Code Linter), command line tools (e.g., hdc), and package management tool (e.g., ohpm). The only exception is the phase of publishing. At the moment, there is no such tool offered for OpenHarmony. It is nonetheless understandable as there is no app market available for hosting OpenHarmony apps yet. We believe such a tool will be provided once a dedicated app market is offered.

It is worth noting that, at the moment, we only conducted a high-level overview and did not check in detail to what extent are the required functions in each phase covered by these tools. For example, there is a tool called *Monkey* in Android that supports random exploration of Android apps, it is not clear to us if the existing toolchains of OpenHarmony provide equivalent functions. As for our future work, we plan to have a more detailed look at these tools and provide the community with a clearer overview of these toolchains.

Table 5. A selected list of OpenHarmony Toolchains.

Tool	SE Phase	Function
DevEco Studio	Development	The recommended integrated development environment for implementing OpenHarmony apps.
Device Manager	Development	This tool provides an interface for developers to manage OpenHarmony devices, including both emulator-based and real-world devices.
Emulator	Development	This tool can set up OpenHarmony emulators (either remotely or locally) that allow developers to install, run, and test their apps on an emulator instead of real-world OpenHarmony devices.
hvigor	Build	The recommended tool for building OpenHarmony source code project to runnable apps.
arkXtest/jsunit	Test	This tool allows developers to run unit tests when implementing OpenHarmony apps.
arkXtest/uitest	Test	This tool allows developers to search and update certain widgets in a given GUI page, which is essential for supporting automated OpenHarmony app testing.
HiLog	Debug	The default tool that is designed to log information such as user operations or system running statuses for the system framework, services, and OpenHarmony apps.
profiler	Debug	This tool provides a visual interface for developers to quickly check the profiling information such as the currently used system and memory resources, including the heap and stack memories of each task.
Code Linter	Code Review	This tool is responsible for grammatically checking the correctness of ArkTS code, which is the default programming language for implementing OpenHarmony apps.
hdc	Other	The OpenHarmony Device Connector tool allows developers to connect their PC-side development machine to a given OpenHarmony device.
ohpm	Other	OpenHarmony Package Manager.

3.2 Existing Datasets

As shown in Section 2, the datasets targeted by our MSE community can be mainly divided into four types: (1) Mobile apps (including both open-sourced[7] and closed-sourced apps[8]), (2) Mobile App Development Framework, (3) Third-party Libraries, (4) App Store Info (including app reviews). We now respectively summarize the current situation of these types of datasets in OpenHarmony, respectively. We further go one step deeper to harvest the relevant datasets, if possible, and make them publicly available to support our fellow researchers in conducting OpenHarmony-related software engineering research.

Table 6. The framework repository comparison between OpenHarmony and Android.

Type	OpenHarmony	Android
Name	OpenHarmony/interface_sdk-js	aosp-mirror/platform_frameworks_base
Platform	Gitee	Github
#. Branches	136	500
#. Tags	34	2,026
#. Forks	1,900	6,300
#. Stars	83	10,600
#. Commits	10,898	946,393
#. Contributors	627	1,399

OpenHarmony Framework. Recall that OpenHarmony is a fully open-sourced system, its app development framework is open-sourced. The framework is the first gate that OpenHarmony apps need to interact with before running into the system. The interaction is mainly through APIs provided by the app development SDK, as part of the OpenHarmony framework. Some of the meta-data of the OpenHarmony framework are shown in Table 6. The current framework is open-sourced at the *interface_sdk-js* repository[9] on Gitee and it currently has 105 branches, 30 tags, 1,400 forks, 57 stars, 7,833 commits, and 627 contributors. As a comparison, the last column of Table 6 shows the meta-data of the Android framework repository, respectively. It is obvious that OpenHarmony has a big step to go in order to catch up with Android, which poses lots of opportunities for our MSE community to mitigate the gap between the OpenHarmony framework and the Android framework.

We further look into the number of APIs offered by the OpenHarmony framework. Since there is no such information directly provided on the web, we decided to write a parser to directly harvest that from the open-source repository. We select the latest version (i.e., OpenHarmony 4.0) and only count the number of functions (including static and non-static functions). In the latest version, there are 10,435 APIs. This number is also significantly smaller than that of the Android framework, which already has over 30,000 APIs in 2018 (i.e., API version 28 [57]). Nonetheless, as illustrated in Fig. 4, the number of APIs (again, any functions are considered) is continuously increasing, showing that the capabilities of OpenHarmony are keeping maturing. We believe as time goes by, such a difference between the APIs of Android and OpenHarmony will be much smaller.

OpenHarmony Apps. One of the most important reasons that make mobile software engineering (especially for the Android community) a longstanding hot topic is due to the existence of a large number of mobile apps [32, 43]. Indeed, there are over 2 million Android apps (there is a similar number for the iOS community) available on the official Google Play store. In the famous AndroZoo dataset [60], there are over 23 million Android

[7]AndroZooOpen: https://github.com/HumaniSELab/AndroZooOpen

[8]AndroZoo: https://androzoo.uni.lu

[9]We remind the readers that the framework and the SDK are not exactly the same as the framework may contain more capabilities that are reserved for system apps while SDK is only supposed to be used by third-party apps. For simplicity, in this work, we will not differentiate this as there is no direct repository provided for hosting the framework code of OpenHarmony.

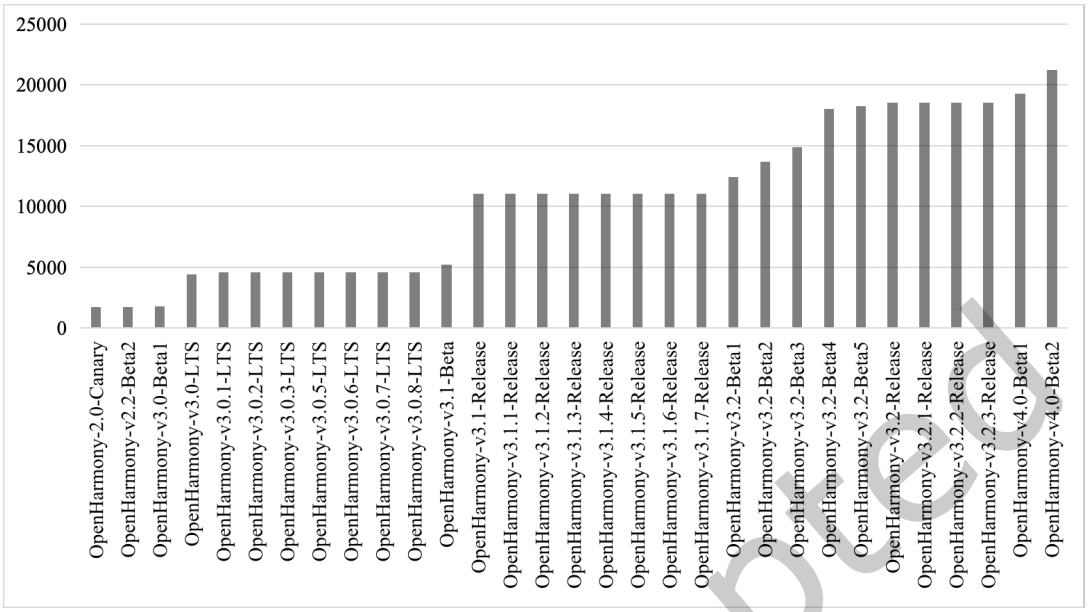

Fig. 4. The evolution of the number of APIs offered by the OpenHarmony framework. The X-axis includes all the tags (ranked based on their released time, the earlier, the former) available in the OpenHarmony repository.

apps collected from various sources (e.g., the official Google Play store and over 10 third-party markets such as PlayDrone, AppChina, etc.) spanning various years. Liu et al. have subsequently harvested the open-sourced Android apps and formed them as a dataset called AndroZooOpen [64]. This dataset is also made publicly available to the software engineering community and has been demonstrated to be useful in supporting Android research tasks. Inspired by this, we hypothesize that OpenHarmony apps will be one of the most important resources for supporting OpenHarmony research. We, therefore, take our initial attempt to harvest existing OpenHarmony apps. Since there is no app market available for OpenHarmony yet, we solely focus on open-sourced OpenHarmony apps. Specifically, we take OpenHarmony as well as HarmonyOS as the search keyword and apply it to two famous cloud-based software version control websites, namely GitHub and Gitee, which are the most famous sites of such in the world and in China, respectively.

Our initial search results in 3,804 repositories[10], for which 910 of them are from GitHub while the remaining 2,894 from Gitee. We remind the readers that these identified repositories may not always be OpenHarmony apps. Therefore, we resort to a Shell script (with manually identified features of OpenHarmony apps considered) to select such repositories that indeed contain OpenHarmony apps. Our experiment has eventually discovered 174 such repositories, with 147 and 27 from Gitee and Github, respectively. To facilitate further research, we have also made this list publicly available on the same site.[11]

Third-party Libraries We remind the readers that OpenHarmony takes a newly introduced language called ArkTS to support app implementations. In this work, we also look at the existing third-party libraries that are available for supporting the development of OpenHarmony apps. Specifically, we would like to understand to what

[10]The full list is made available on GitHub (https://github.com/SMAT-Lab/SE4OpenHarmony).

[11]Our further investigation finds that most of these apps are not comprehensive ones (i.e., might be toy apps or demonstrating the usability of certain libraries). We hence commit to keep updating this list toward forming a more useful dataset for supporting OpenHarmony-based software engineering research.

extent are ArkTS-based libraries available in our community and what are they designed for. In OpenHarmony, the official team has introduced a tool called *ohpm* (as also shown in Table 5) for managing all the third-party libraries designed for developing OpenHarmony apps. In the current central registry[12], there are already 96 libraries and the number is growing. The functions of these libraries can be divided into 10 categories. Table 7 enumerates some of the representative ones for each category. Generally, we show one or two libraries for each category. The two are randomly chosen if there are more libraries available for the given category.

Table 7. A sample list of OpenHarmony's third-party libraries (available in OpenHarmony's central registry).

Category	Count	Repo	
UI	2	@ohos/pulltorefresh	Pull-to-refresh and pull-up loading component
		@ohos/mpchart	Support the implementation of various types of charts such as Pie chart, Candle chart, etc.
Animation	2	@ohos/lottie	The animation library for OpenHarmony. Similar to Java's lottie, Android-ViewAnimations, and Leonids libraries.
		@ohos/svg	SVG-formatted image parser and rendering library.
Network	1	@ohos/axios	The promise-based HTTP Client implementation library for OpenHarmony.
Image	2	@ohos/imageknife	An efficient, lightweight, and simple image loading cache library
		@ohos/xmlgraphicsbatik	For working with images in SVG format
Multimedia	1	@ohos/ijkplayer	FFmpeg-based video player
Data Storage	2	@ohos/disklrucache	Support cache functions for accessing disks
		@ohos/mmkv	A lightweight key-value storage framework
Event	2	@ohos/mqtt	Support MQTT-based actions such as message subscription
		@ohos/liveeventbus	Support inter-process and inter-app message broadcast
Security	1	@ohos/crypto-js	Support the implementation of cryptographic functions such as MD5, SHA256, etc.
Utility	2	@ohos/zxing	Support read or generate QR Code for OpenHarmony
		@ohos/pinyin4js	Translating Chinese characters to pinyin
Other	2	@ohos/arouteronactivityresult	Support message transmission when performing inter-page or inter-app communications.
		@ohos/coap	Support Constrained Application Protocol (CoAP) capabilities.

Furthermore, as mentioned previously, ArkTS is not entirely new. It actually extends Typescript, which has been a popular programming language for more than 10 years. Typescript is Javascript with syntax for types, i.e., adding static typing with optional type annotations to Javascript. Theoretically, existing Typescript code (as well as Javascript code) can be directly reused for developing OpenHarmony apps. Those Typescript/Javascript implementations could be regarded as third-party libraries as well. By taking Typescript and Javascript as the search keywords, Github returns 513,000 and 1.7 million repositories for Typescript and Javascript, respectively. Such a large number of repositories (despite not all of them being code-related repositories) indicates that there are already a lot of potential third-party libraries available for OpenHarmony.[13] Those libraries could be leveraged (either directly or with additional efforts contributed by our fellow researchers) to facilitate the development of OpenHarmony apps and its broad ecosystem.

App Store Info. Our software engineering researchers have leveraged app store info (such as the app's author information, description, user rating, user reviews, etc.) to support various studies. For example, Gorla et al. [39] have leveraged the app's description to check against the app's behaviour. Obie et al. [78] have leveraged the

[12]https://ohpm.openharmony.cn

[13]We hypothesize that this is one of the major reasons why ArkTS is proposed as the default programming language for developing OpenHarmony apps.

app's review data to investigate the violation of honesty in mobile apps. To the best of our knowledge, there is barely any app store hosting OpenHarmony apps at the moment. Therefore, there is no such dataset that can be collected so far. Nonetheless, the OpenHarmony version of a given app will also share much of such information as that available in Android or iOS. This information could also be helpful when mining OpenHarmony-specific app store information.

3.3 Existing OpenHarmony Research

As our initial attempt towards building the research roadmap for guiding our software engineering researchers to contribute to OpenHarmony, we start by conducting a tertiary study about OpenHarmony. Our method is straightforward. We use *OpenHarmony* and *HarmonyOS* as the search keywords and we apply them separately to search for relevant publications on both *Google Scholar* and *DBLP*, respectively. At this step, when applied to Google Scholar, we will only consider the top 100 results. Table 8 enumerates the list of OpenHarmony-related publications. In total, we only found 8 papers among which only one (i.e., the one published at the APWeb conference) can be found on DBLP, while all of them can be found on Google Scholar. At this step, we only consider a paper relevant if and only if it directly contributes to the OpenHarmony project or if it takes OpenHarmony as its dataset to evaluate their approaches. There are several other papers that are not included in this review although they do involve OpenHarmony/HarmonyOS systems. They are excluded because they do not contribute anything to OpenHarmony as they only involve running their approaches on OpenHarmony/HarmonyOS systems. For example, the work proposed by Qiu et al. [87] is not included in this paper because it only leverages HarmonyOS to support their model implementation about supporting distributed user interfaces to be dynamically configured on multiple IoT devices based on user preferences.

Table 8. The list of OpenHarmony-related primary publications.

Year	Title	Relevance	Venue	CORE-Rank
2023	CiD4OhOs: A Solution to HarmonyOS Compatibility Issues	API-induced compatibility issues	Industry Challenge Track of ASE	A
2023	HiLog: A High Performance Log System of OpenHarmony	Targeted OpenHarmony's log system	Journal of Software	-
2023	Design and Implementation of HiLog, the high-performance log system of OpenHarmony	Targeted OpenHarmony's log system	Journal of Software	-
2023	Breaking the Trust Circle in HarmonyOS by Chaining Multiple Vulnerabilities	Investigated the security of HarmonyOS's trust circle service	ACCTCS	-
2023	Unveiling the Landscape of Operating System Vulnerabilities	Studied HarmonyOS's vulnerabilities	Future Internet	-
2022	A Deep Looking at the Code Changes in OpenHarmony	Studied OpenHarmony's code changes	APWeb	B
2022	Cross Platform API Mappings based on API Documentation Graphs	Studied HarmonyOS's API documentation	QRS	B
2021	SparrowHawk: Memory Safety Flaw Detection via Data-driven Source Code Annotation	Applied to detect vulnerabilities in OpenHarmony	Inscrypt	National

As shown in Table 8, there are only eight OpenHarmony-related papers published in the community. The efforts could be neglected if compared to those for Android, where there are over 7,000 papers published as recorded in DBLP (searching by taking Android as the keyword). This evidence confirms our previous argument that there is still a huge gap between OpenHarmony and Android. This, however, also demonstrates that there are huge opportunities open for our community. Ideally, the research methods applied to Android or iOS could also be applied to OpenHarmony. Despite there being only eight papers published, it is motivating to find that the number of relevant papers keeps growing. The venues where the current papers are published are generally not in reputed journals or conferences. Indeed, among the eight papers, only four of them are published at venues

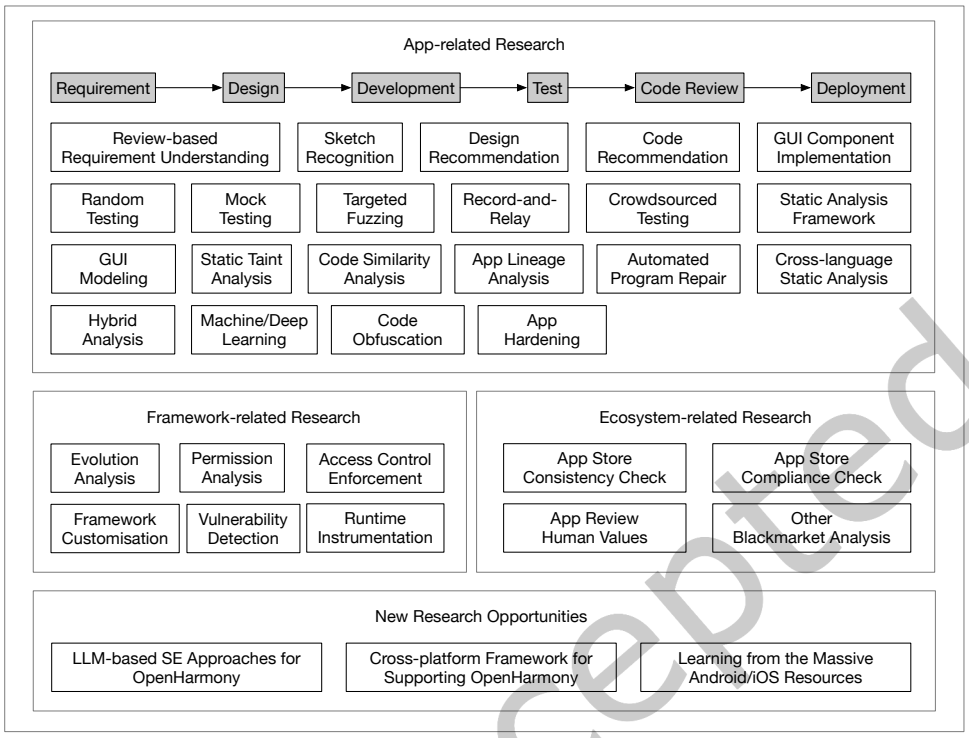

Fig. 5. Overview of Research Gaps between OpenHarmony and MSE.

recorded by CORE and only three of them are ranked. We would hope that our community could spend more effort in developing software engineering approaches for OpenHarmony and publish more papers at mainstream venues.

4 A Research Roadmap for OpenHarmony

As our initial attempt to prompt software engineering research for OpenHarmony, we now present the preliminary research roadmap by summarizing the research gaps between Android/iOS and OpenHarmony. When detailing the gaps, we also present example works that we believe should be proposed for OpenHarmony. We hope these works could be contributed by our fellow researchers so as to fill the aforementioned gaps, making OpenHarmony a popular mobile platform and a popular research topic in the mobile software engineering field.

Specifically, at a high-level matter, we summarize a set of research gaps between OpenHarmony with the existing software engineering works as follows:

(1) As discussed in this work, based on our tertiary study, we found that there are many papers that propose static program analysis approaches for Android apps. Many of those approaches (such as FlowDroid, IccTA, DroidRA, etc.) leverage a common static analysis framework called Soot. In OpenHarmony, we have found that the community has also started to build such a common static analysis framework, aiming at facilitating the implementation of more OpenHarmony-specific software engineering approaches [80].
(2) Except for software engineering works based on static program analysis, there are also many approaches proposed based on dynamic app tests. To facilitate automated app testing approaches, the OpenHarmony

platform has provided the uitest tool [83] for helping users quickly test the UI pages of given apps, as well as the Wukong tool for testing the stability of the OS [81].

(3) One of the big problems faced by Android is its security, many software engineering works have been proposed to secure Android apps. Similar to that, the OpenHarmony community also feels security will be a big issue and thereby has created a Security-SIG dedicated to handling security threats [82].

(4) Similar to Android, which has encountered significant compatibility issues when adopted by downstream companies such as Samsung or Xiaomi. Similarly, since OpenHarmony, as an open-sourced mobile system, would be also adopted by other downstream companies, it may also suffer from similar compatibility issues, as that have been faced by Android. To this end, compatibility tests have been adopted by the OpenHarmony community to mitigate this [79].

In the following sections, we will break down these gaps further, exploring specific areas where existing works can converge to enhance the OpenHarmony ecosystem.

4.1 Gaps in App-related Research

As highlighted in Section 2, the majority of MSE studies focus on mobile apps. At the beginning of this section, we first summarize some of the representative works that propose software engineering techniques to support their studies. Specifically, we summarize them based on the general software development processes, including Requirement, Design, Development, Test, Code Review, and Deployment. As expected, there are fewer works that target the phases before app development. Indeed, most of the works are proposed to examine mobile apps once they are developed.

(1) **[Requirement] Mining User Reviews for Requirement Analysis.** Since it is generally not possible to obtain the original requirements of mobile apps (e.g., what functions to offer and how should they interact with users), which are often considered confidential, the research community mainly focuses on mining user reviews for requirement understanding. Here, user reviews can be collected either through actual interviews or through user comments made to the app's release page on the app store. Fortunately, such efforts can directly be leveraged to improve OpenHarmony apps as the identified requirements are often independent of mobile platforms. Nevertheless, the proposed techniques could be also leveraged to mine user reviews that are specifically made for OpenHarmony apps.
⟶**Representative Works:** Chen et al. [20] argue that it is possible to dig out user needs and preferences by analyzing user online comments, which can subsequently benefit app developers to make accurate market positioning and thereby increase the volume of app downloads. By using a set of NLP techniques such as semantic analysis, and word frequency analysis, the authors demonstrate the possibility of obtaining useful requirements. Similarly, Palomba et al. [84] propose to support the evolution of mobile apps via crowdsourcing user reviews. By surveying 73 developers, they have found that over 75% of developers will take user reviews into consideration when updating their apps and such updates are often rewarded in terms of significant increases of user ratings.

(2) **[Design] Sketch Recognizing** App designers often use sketches to quickly draw the app's user interfaces so as to accelerate the iterative design process when designing apps. Such sketches, however, cannot be directly used to build a prototype app that can be immediately tested to collect user feedback. To bridge the gap, researchers have proposed techniques to automatically recognise sketches and subsequently transform them into UI components. In this way, app developers can focus on designing the user experience rather than building the prototypes with various tools. Such approaches could be extremely beneficial to OpenHarmony developers when designing their apps.
⟶**Representative Works:** Kim et al. [47] have presented to the community an approach to identify UI widgets of mobile apps directly from sketch images using geometric and text analysis features. The

extraction of graphic elements such as text or shapes from the input sketch image using the Optical Character Recognition (OCR) technique and edge detection. Similarly, Li et al. [61] have proposed to the community a sketch-based prototyping tool called Xketch for accelerating mobile app design processes. They have demonstrated that Xketch is indeed useful and can benefit app developers in designing apps quickly on their tablets.

(3) **[Design] Visual Search for Recommending Design Examples.** Since it is non-trivial to design a beautiful user interface from scratch, developers often resort to relative UI design examples to gain inspiration and compare design alternatives. However, finding such design examples is challenging as existing search systems only support text-based queries. To mitigate this, our community has proposed to conduct a visual search, which takes as input a UI design image and outputs visually similar designs. Since visual search is independent of mobile platforms, such efforts can directly be leveraged to benefit the OpenHarmony community as well. Nevertheless, OpenHarmony apps may have specific preferences in their UI pages, there is also a need to invent dedicated visual search systems to support the design of OpenHarmony apps.
⟶**Representative Works:** Bunian et al. [18] have proposed to the community a visual search system, which includes an object-detection-based image retrieval framework that models the UI context and hierarchical structure. Based on a large-scale UI dataset, the authors have shown that their visual search framework can achieve high performance in querying similar UI designs.

(4) **[Development] Code Recommendation.** Mobile apps are developed based on an official SDK with thousands of APIs, and there are hundreds of thousands of APIs available in the wild through the so-called third-party libraries. There is hence a strong need to automatically recommend appropriate APIs (or libraries) for developers to choose when they implement their apps. Furthermore, libraries have been demonstrated to be extremely useful for facilitating app development as they provide lots of existing function implementations that are reusable and are often high-quality (e.g., being already validated by their various usages). It is not uncommon to encourage developers to leverage third-party libraries for implementing OpenHarmony apps. As the number of available libraries keeps growing, it is non-trivial for developers to search for the appropriate libraries. Therefore, there is a strong need to automatically recommend the required libraries for OpenHarmony app developers.
⟶**Representative Work:** Zhao et al. [129] have presented to the community a prototype tool called APIMatchmaker that automatically matches the correct APIs for supporting the development of Android apps. The recommended APIs are learned from other Android apps that are deemed similar to the one under development.

(5) **[Development] GUI Component Implementation.** Mobile apps involve lots of icons. To maintain the same look and feel, similar GUI components (icons or animations) across different mobile apps often reserve similar functionalities. Therefore, it is possible to learn the semantics behind popular GUI components and subsequently recommend code implementations to developers when relevant GUI components are used.
⟶**Representative Work:** Zhao et al. [128] have proposed an approach called Icon2Code that leverages an intelligent recommendation system for helping app developers efficiently and effectively implement the callback methods of Android icons. The recommendation system is built based on a large-scale dataset that contains mappings from icons to their code implementations. Similarly, Wang et al. [108] have proposed an approach to recommend APIs for implementing Android UI animations. This approach constructs a database containing mappings between UI animations in GIF/video format and their corresponding APIs and subsequently leverages it to achieve the recommendation.

(6) **[Test] Random Testing (Test Case Generation).** Like any software, mobile apps must be thoroughly tested before release, and this is equally true for OpenHarmony. Two key scenarios require test case generation to ensure app reliability: unit testing, which verifies the correctness of specific functions, and

input generation for apps running on mobile operating systems. These needs are also relevant to the OpenHarmony community. Random testing is widely recognized for its ease of use and scalability in generating test cases to explore mobile apps. Researchers have demonstrated that Monkey, a simple random testing tool for Android apps, is surprisingly effective, often achieving higher code coverage than more sophisticated tools. Similarly, OpenHarmony can benefit from adopting random testing approaches, which serve as a foundation for developing advanced app testing tools.

⟶**Representative Work:** Amalfitano et al. [6] have presented to the community a research prototype named AndroidRipper, which embeds an automated technique that tests Android apps via their Graphical User Interface (by automatically explores the app's GUI with the aim of exercising the application in a structured manner). Existing experimental results show that AndroidRipper outperforms the random testing approach, being capable of detecting severe and previously unknown faults in open-source Android apps.

(7) **[Test] Mock Testing.** Performing unit tests for mobile apps, including OpenHarmony apps, is non-trivial. Indeed, certain functions under testing require the context that is a part of the app's lifecycle or the system. This context information is only available when the app is running on the mobile system, which is contradictory to the fact that unit tests do not expect to have the apps actually run on mobile devices.

⟶**Representative Work:** There are several well-known frameworks such as Mockito in MSE that are provided by practitioners to support mock unit testing. Similar frameworks are highly demanded by the OpenHarmony community as well. On the research side, Beresford et al. [15] have proposed to the community a novel approach called MockDroid that allows a user to 'mock' an app's access to a resource. The resource is subsequently reported as empty or unavailable whenever the app requests it. Their work is demonstrated to be useful for testing mobile apps w.r.t. their tolerance to resource failures.

(8) **[Test] Targeted Fuzzing.** Modern mobile apps run on touch-sensitive displays with numerous GUI pages, each involving various lifecycle methods and widgets linked to callback methods. This complexity makes achieving highly efficient random testing challenging. To address this, researchers have proposed targeted fuzzing, which generates test inputs to guide the app toward specific states. Given the GUI-intensive nature of OpenHarmony apps, the limitations of random testing also apply. Thus, there is a strong need to develop targeted fuzzing approaches to effectively test OpenHarmony apps.

⟶**Representative Work:** Rasthofer et al. [91] present to the community a targeted fuzzing approach, namely FuzzDroid, for automatically generating an Android execution environment where an app exposes malicious behaviour. This objective is achieved by combining an extensible set of static and dynamic analyses through a search-based algorithm that steers the app toward a configurable target location.

(9) **[Test] Record-and-Replay.** After release, mobile apps must run on various devices with differing framework versions and screen sizes, including customized frameworks. To ensure consistent behavior across devices, researchers propose Record-and-Replay testing, which records a test scenario on one device and replays it on others to verify identical results. Given OpenHarmony's "1+8+N" strategy—supporting one main device (e.g., smartphone), eight key devices (e.g., TV, Smartwatch, Pad, PC), and numerous user-customized devices—Record-and-Replay testing is crucial for ensuring the strategy's success.

⟶**Representative Work:** Gomez et al. [38] present a prototype tool called RERAN that achieves timing- and touch-sensitive record-and-replay for Android. RERAN attempts to directly capture the low-level event stream on the phone and replay it later on with microsecond accuracy. Since mobile apps may be run on different devices with diverse screen sizes, a record-and-replay tool may be applied to apps that could have different GUI layouts on different devices.

(10) **[Test] Crowdsourced Testing.** Automated app testing cannot achieve 100% coverage and hence user commitments are always needed in order to ensure the quality of mobile apps. However, manually exploring an app in a comprehensive way is difficult and time-consuming. To alleviate that, researchers have proposed

to leverage crowdsourcing efforts to achieve the aforementioned testing purpose. Indeed, crowdsourced testing provides a promising way to conduct large-scale and user-oriented testing scenarios. Such an approach could be also leveraged to comprehensively test OpenHarmony apps.

⟶**Representative Work:** Ge et al. [36] find that most crowdsourced app testing is of low quality as crowd workers are often unfamiliar with the app under test and do not know which part of the app should be tested. To fill this gap, the authors propose to construct an Annotated Window Transition Graph (AWTG) model for the app under test by merging dynamic and static analysis results and subsequently leverage the AWTG model to implement a testing assistance pipeline that offers test task extraction, test task recommendation, and test task guidance for crowd workers. Recently, Sun et al. [104] present to the community a lightweight approach that aims to achieve fully automated crowdsourced app testing by only dispatching the app's partial code for crowdsourced execution. The experimental results involving tests of API-related code only (of real-world apps) show that their approach is useful (as demonstrated by being able to find many API-induced compatibility issues) and welcome in practice.

(11) **[Code Review] Static Analysis Framework.** Static analysis is a fundamental technical that has been frequently applied to resolve various Android app analysis problems. Such solutions are often implemented based on the so-called static analysis frameworks that offer implementations to core static analysis functions such as control-flow graph construction, call graph constructions, etc. OpenHarmony takes a new program language called ArkTS to develop its apps. Therefore, an ArkTS-specific static analysis framework is required to support the implementation of other purpose-oriented static analysis approaches (e.g., vulnerability detection).

⟶**Representative Work:** Soot [54] is one of the most popular static analysis frameworks that are capable of analyzing Android apps. Soot is initially designed for Java program analysis and is further extended for Android apps (which are written in Java) thanks to the Dexpler module contributed by Bartel et al. [14]. Another popular static analysis framework should be the one named WALA [95], which is developed and maintained by IBM. In Android, both Soot and WALA have been recurrently adopted by our fellow researchers to support the implementation of static analysis approaches.

(12) **[Code Review] GUI Modeling.** Android apps rely on complex graphical user interfaces (GUIs), challenging static analysis. GUI pages often contain numerous UI widgets, arranged via various layout strategies, each handling diverse user events (e.g., clicks). Android GUIs further complicate analysis as they can be defined both statically (XML) and dynamically (Java code). Consequently, specialized methods are required to model GUIs and analyze app behavior effectively.

⟶**Representative Work:** ArchiDroid [68] statically analyzes the transition relationship among activities of apps and constructs the activity transition graph. It also models the activity semantic and graph structure information via graph convolution network to automatically predict transitions between activities and augment the activity transition graph built by static analysis. Besides static analysis-based approaches, SceneDroid [125] explores activities and extracts the GUI scenes by a series of dynamic analysis techniques, and then presents the GUI scenes as a scene transition graph to model the GUI of apps.

(13) **[Code Review] Static Taint Analysis (for Detecting Privacy Leaks).** One of the most popular usages of static analysis is to perform static taint analysis for pinpointing sensitive data flows (also known as privacy leaks). Static taint analysis works by first coloring some variables that contain sensitive data such as the user's phone number and then tracking their flows in the code. A sensitive data flow is considered detected if such coloured data eventually flows to sensitive operations (e.g., sending the coloured data outside the device via SMS). OpenHarmony apps will be run on mobile devices and hence will have similar requirements. Therefore, it is also essential ed to invent static taint analysis approaches for examining OpenHarmony apps.

⟶**Representative Work:** Arzt et al. [8] have presented to the MSE community an open-source tool called FlowDroid, which performs context-, flow-, field-, and object-sensitive and lifecycle-aware taint analysis for Android apps. The authors further provide on-demand algorithms for FlowDroid to achieve high efficiency and precision at the same time.

(14) **[Code Review] Code Similarity Analysis.** Code similarity analysis is another common application of static analysis that has also been recurrently adopted by developers to achieve various functions, e.g., to detect code clones, the usage of third-party libraries, and repackaged (or piggybacked) apps. Code similarity analysis is also essential to understand the difference between two code snippets, including the two timestamped versions of the same code snippet. Such a difference can then be leveraged to support the implementation of various software engineering tasks such as automatically generating commit messages or inferring patches to given code defects, etc.
⟶**Representative Work:** Russell et al. [22] have presented to the MSE community a prototype tool called AnDarwin for detecting semantically similar Android apps. AnDarwin leverages a clustering-based approach, for which it attempts to cluster similar apps into the same group based on semantic information extracted from the apps' code.

(15) **[Code Review] App Lineage Analysis.** Due to the fast evolution of the OS framework as well as the requirement to fix bugs or add new features, mobile apps are continuously updated by their developers (often over app stores). Such updates will lead to a series of releases of the same app, which is referred to by the community as app lineages. Because these app lineages have recorded all the app changes, our fellow researchers have proposed to mine them[14] to learn why the mobile apps updated. Similar approaches could also be applied to OpenHarmony, e.g., to mine knowledge for updating (or fixing) existing apps.
⟶**Representative Work:** Gao et al. [34] presents an experimental study about the evolution of Android app vulnerabilities. They first define the term "app lineage" (i.e., the series of a given app's historical versions). Then, they collect a dataset of app lineages and subsequently leverage it to understand the vulnerability evolution by mining the updates between an app's two consecutive versions. Their empirical study has revealed various interesting findings. The authors further conduct another work to mine app lineages for understanding the evolution of Android app complexities [33]. Their experimental results reveal a controversial finding where app developers might not really be aware of controlling the complexity of their apps.

(16) **[Code Review] Automated Program Repair.** Automated Program Repair (APR) has been a hot topic in the software engineering community for years. The idea of APR is for computers to automatically produce source code-level patches for bugs and vulnerabilities. Our fellow researchers have also attempted to invent techniques to automatically repair mobile apps. We argue that such techniques should also be explored to target OpenHarmony apps.
⟶**Representative Work:** Marginean et al. [72] present an industry tool called SapFix that achieves end-to-end fault fixing, from test case design to deployed repairs in production code. SapFix achieves its purpose by combining a number of different techniques, including mutation testing, search-based software testing, and fault localization. Zhao et al. [127] have presented to the community another prototype tool called RepairDroid, which aims at automatically repairing compatibility issues directly in published Android apps (at the bytecode level). To support flexible repair, the authors have introduced a generic app patch description language that allows users to create fix templates using IR code.

(17) **[Code Review] Cross-language Static Analysis.** Mobile apps are not always written in a single programming language. Indeed, there are various apps that are implemented in multiple languages. For example, the module requiring high performance in Android apps could be written in C or C++ while the main part

[14]Researchers have to focus on the app's released versions because it is often not possible to obtain its source code.

is still written in Java, which is the default language to implement Android apps. As another example, for such Android apps that leverage web-related components, certain functions could be written in Javascript, in order to supplement the main functions written in Java. In order to properly analyze these apps involving multiple programming languages, we argue that there is a need to conduct cross-language static analysis, for which the data-flow analysis should propagate variables from one language to another.
⟶**Representative Work:** Wei et al. [110] and Zhou et al. [130] demonstrate that it is important to support inter-language static analysis in order to support security vetting of Android apps. To do so, Samhi et al. [94] present to the community a prototype tool called Jucify that aims to unify Android code (between Java and C/C++) to support static analysis. Their work is able to build a comprehensive call graph across all the methods written in the app, no matter they are written in Java or C/C++. Xue et al. [114] have also invented a prototype tool called NDroid for tracking information flows across multiple Android contexts, including the analysis of native code in Android apps [131].

(18) **[Code Review] Hybrid Analysis.** As discussed previously, both testing (also known as dynamic analysis) and static analysis techniques are recurrently adopted by our fellow researchers to dissect mobile apps. However, both of these two techniques are known to have drawbacks, e.g., testing approaches suffer from code coverage problems that eventually lead to false negative results, meanwhile, static analysis is known to likely yield false positive results. To mitigate this, our fellow researchers have proposed to combine these two approaches to conduct the so-called hybrid analysis of mobile apps. We believe there is also a need to invent hybrid approaches for analyzing OpenHarmony apps.
⟶**Representative Work:** Wang et al. [107] present an automated hybrid analysis of Android malware through augmenting fuzzing with forced execution. They propose an approach called DirectDroid, which aims to trigger hidden malicious behaviour by bypassing some related checks when adopting fuzzing to feed the necessary program input. Spreitzenbarth et al. [101] have developed another hybrid analysis approach called Mobile-Sandbox, for which static analysis is leveraged to reach higher code coverage during dynamic analysis (i.e., app testing).

(19) **[Code Review] Machine/Deep Learning.** Machine Learning has become one of the most popular techniques that are frequently adopted by our fellow researchers for reviewing apps' logic code. Indeed, a lot of research efforts are spent to find the best feature set that could closely represent the app's behaviour. Such a feature set is then leveraged to support two types of machine learning approaches: supervised learning and unsupervised learning. Supervised learning requires knowing the labels of the training dataset, e.g., it is essential to collect a set of known malware in order to train a malware predictor. On the contrary, unsupervised learning does not need to know the labels of the dataset. This type of approach is often used to cluster similar samples into the same group. When deep learning is concerned, feature engineering is no longer needed.
⟶**Representative Work:** Liu et al. [66] have recently conducted a systematic literature review about deep learning approaches applied to defend Android malware. The authors have surveyed papers published from 2014 to 2021 and have located 132 closely related papers. The authors find that static analysis is the most used technique to obtain features from Android apps and there are 13 works that achieve malware classification by directly encoding the raw bytecode of Android apps into feature vectors. Machine learning is not only applied to dissect malware but is also used for resolving other software engineering tasks. For example, Rasthofer et al. [90] have presented to the community a machine learning-based approach for classifying and categorizing sources and sinks in Android, which can then be leveraged to support taint analysis of Android apps, so as to detect privacy leaks.

(20) **[Deployment] Code Obfuscation.** Because of the nature of mobile devices, mobile apps need to be downloaded to the devices before installation. This, unfortunately, makes it possible for attackers to directly access the mobile apps. Even worse, the attackers might be able to directly access the code implementations

of the apps if reverse engineering techniques are applied. To prevent attackers from easily understanding the code, the MSE community has adopted the practice of performing code obfuscation before assembling the app code to a release version. Since OpenHarmony apps need to be installed on users' devices, it is also essential to invent code obfuscation techniques to prevent OpenHarmony apps from being exploited by attackers.

⟶**Representative Work:** Aonzo et al. [7] has developed an open-source black-box obfuscation tool for Android apps. The authors named their approach Obfuscapk and have designed a modular architecture for users to straightforwardly extend so as to support the implementation of new obfuscation strategies. Dong et al. [27] conduct a large-scale empirical study of Android obfuscation techniques, with the hope of better understanding the usage of obfuscation. The authors have specifically looked into four popular obfuscation approaches: identifier renaming, string encryption, Java reflection, and packing, leading to various findings that could help developers select the most suitable obfuscation approach.

(21) **[Deployment] App Hardening.** Code obfuscation is a useful technique to prevent attackers from easily understanding the code. It is nonetheless not possible to prevent attackers from obtaining the code. With the help of deobfuscation approaches, attackers could still understand the implementation details. To prevent that from happening, the MSE community further introduced to the community the so-called app hardening technique, which aims to make it difficult to extract code implementation from the apps (e.g., will stop reverse engineering tools from disassembling released apps).

⟶**Representative Work:** Russello et al. [92] present to the MSE community a policy-based framework called FireDroid that enforces security policies without modifying Android OS or the actual applications. FireDroid includes a novel mechanism to attach, monitor, and enforce policies for any process spawned by the Android's mother process Zygote. With that, FireDroid can be applied to block OS and app vulnerabilities, hardening security on Android phones. Zhang et al. [126] have conducted the first systematic investigation on Android packing services toward understanding the major techniques used by state-of-the-art packing services and their effects on apps. They further find that the protection given by those packing services is not reliable, i.e., the Dex can be recovered. To demonstrate that, the authors have designed and implemented a prototype tool called DexHunter for extracting Dex files from packed Android apps. Following that, Xue et al. [115–117] have gone steps further to achieve unpacking through various methods, e.g., Hardware-assisted approach, VM-based approach, etc.

4.2 Gap in OS Framework-related Research

As highlighted in Fig. **??**, the OS framework is the layer that connects the apps with the system capabilities. It provides all the necessary capabilities (including all the APIs offered by the SDK) to support apps running on mobile devices. Since this part is closely related to apps, it has also been a frequent topic targeted by our SE researchers. We now summarize the representative ones.

(20) **[Static] Evolution Analysis.** Like what has been done for mobile apps, our fellow researchers have also proposed approaches to study the evolution of OS frameworks. They have shown that understanding the evolution of the framework could provide useful information for the mobile community. However, unlike mobile apps, the studies related to the evolution of OS framework are mainly based on source code as the framework (mainly Android framework) is open-sourced. Since OpenHarmony's framework is also open-sourced, such techniques applied to study the evolution of the Android framework could be also applied to OpenHarmony.

⟶**Representative Work:** Li et al. [57] have proposed to study the evolution of the Android framework to characterize deprecated APIs. Their empirical study has revealed various interesting findings including the inconsistency among the API's implementation, its comments, and annotations. They have also found that

the Android framework includes a lot of inaccessible APIs that are not designed to be invoked by client apps but have actually been accessed in practice [59]. As argued by Liu et al. [63], by looking into the evolution of Android APIs, we could find the silently evolved APIs that could eventually lead to indiscoverable compatibility issues [104] as the API's implementation is updated during the evolution while its comment remained the same.

(21) **[Static] Permission Analysis.** The Android permission system, a key security mechanism, has been extensively studied by the software engineering community. Ideally, apps should declare only the permissions they require, but the lack of a clear mapping between permissions and the APIs provided to developers often leads to over-declaration. This enlarges the attack surface, making apps more vulnerable. Researchers have addressed this by analyzing framework code to build permission-to-API mappings, enabling finer-grained permission analysis. Since OpenHarmony also employs a permission system to secure apps, it is likely to face similar challenges as Android. Therefore, conducting analogous research on OpenHarmony is crucial to identify weaknesses, ensure proper permission use, and mitigate potential security risks.
⟶**Representative Work:** Au et al. [9] present to the community a prototype tool called PScout that automatically extracts the permission specification from the Android OS source code (i.e., over a million lines of code) using static analysis. Their approach has resolved several challenges including the one to take into account permission enforcement due to Android's use of IPC and Android's diverse permission-checking mechanisms. Bartel et al. [13] have conducted a similar study by leveraging static analysis for extracting permission checks from the Android framework. Their approach is designed to be field-sensitive with an advanced class-hierarchy analysis strategy and uses novel domain-specific optimizations dedicated to Android.

(22) **[Static] Access Control Enforcement.** Security is not only the biggest problem in mobile apps, it is also one of the biggest problems in the OS framework side. To ensure the security of the system, the OS framework often relies on access control mechanisms to achieve the purpose. However, such access control mechanisms could be bypassed by malware so as to achieve unauthorized security-sensitive operations. Therefore, there is a need to enforce the access control function being properly applied.
⟶**Representative Work:** Zhou et al. [132] have presented to the community a prototype tool called IAceFinder that aims to extract and contrast the access control enforced in the Java and native contexts of Android and subsequently to discover cross-context inconsistencies, as a major means to stop access control functions from being bypassed. The authors have applied their approach to analyzing 14 open-source Android OS frameworks (i.e., ROMs), from which they are able to disclose 23 inconsistencies that can be abused by attackers to compromise the device.

(23) **[Static] Framework Customization.** Due to the openness of Android and the requirement to provide vendor-specific user experience, the Android framework has been recurrently customized by smartphone vendors. For example, Xiaomi has done that and named the customized version MIUI. Similarly, Huawei has released EMUI to feature a more personalized user experience when using Huawei phones. Unfortunately, such a wide range of customizations has introduced significant compatibility issues to the community, making it difficult for app developers to implement an app that is compatible with all the available mobile devices. Our SE researchers have hence proposed approaches to mine the difference between the customized frameworks so as to mitigate the compatibility issues in the mobile community. As an open-source system, OpenHarmony could face similar problems. Therefore, there is also a need to spend research efforts to control the customization and thereby keep such problems from happening in OpenHarmony.
⟶**Representative Work:** Liu et al. [62] have conducted an empirical study to understand whether customized Android frameworks keep pace with the official Android. They have looked at the evolution of eight downstream frameworks (e.g., AOKP, AOSPA, LineageOS, SlimROMs, etc.) and discovered various interesting findings (e.g., Downstream projects perform merge operations only for a small portion of all

the version releases in the upstream project and most of the downstream projects take more than 20 days to bring changes from their corresponding upstream projects). The authors further look at the differences among the customized frameworks (including the ones modified by popular technical companies such as Xiaomi and Huawei) and find that this customization has led to serious compatibility issues (also known as the fragmentation problem) in the Android community [65]. This result strongly suggests that more efforts are required to ensure framework customization is properly handled and managed.

(24) **[Static/Dynamic] Vulnerability Detection.** Due to the complexity and huge codebase of the Android system, vulnerable implementations commonly exist in different aspects of the Android framework. There is hence a need to continuously scan for vulnerabilities so as to improve the system's security. Our fellow researchers have hence proposed various approaches to achieve that, either statically or dynamically. Note that mobile frameworks are often developed with multiple programming languages, vulnerability detection approaches are hence required to support cross-language analyses.
⟶**Representative Work:** Luo et al. [70] have proposed a tool called CENTAUR that discovers the vulnerable interfaces of Android system services that can be exploited by malicious apps to steal private data. In detail, CENTAUR leverages symbolic execution and taint analysis to monitor the variables in the Android framework, which can be compromised by malicious apps to steal private data. In dynamic analysis, Liu et al. [132] proposed an approach called FANS that employs fuzzing techniques to detect vulnerable system services. It statically analyzes the data structure of each parameter of the interfaces of system services and then randomly generates arguments to drive the execution of interfaces for triggering vulnerabilities in system services.

(25) **[Dynamic] Runtime Instrumentation.** As not all issues can be resolved statically, researchers have explored dynamic analysis of frameworks, such as controlling framework execution. A notable approach involves instrumenting the framework by adding hook methods to specific functions. At runtime, these hooks provide valuable runtime information, aiding in understanding the framework's behavior and that of the apps running on it. This technique should also be made available to the OpenHarmony community to enable advanced framework and app analysis.
⟶**Representative Work:** One of the most famous runtime instrumentation approaches in Android is the Xposed framework, which allows developers to install little programs (called modules) to Android devices to customize their look and functionality. On the research side, Costamagna et al. [21] present a similar approach called ARTDroid that supports virtual method hooking on Android ART runtime. As another example, the most representative work related to runtime instrumentation is the one proposed by Enck et al. [29], who have presented to the MSE community one of the first approaches targeting runtime instrumentation in Android. They have implemented an information-tracking system called TaintDroid, aiming to achieve real-time privacy monitoring on smartphones. The runtime instrumentation of TaintDroid is enabled by leveraging Android's virtualized execution environment.

4.3 Gaps in Ecosystem-related Research

Except for the aforementioned research studies related to mobile apps and frameworks, there are also a significant number of studies focusing on the other aspects of MSE, which we refer to in this work as ecosystem-related studies. We now discuss some of the representative ones.

(24) **[App Store] Consistency Check.** App stores, such as Google Play and the Apple Store, have become integral to modern life, serving as centralized repositories for discovering, purchasing, installing, and managing apps. They record extensive app metadata, provided either by authors (e.g., app name, description) or collected by the platform (e.g., user ratings), which aids users in app discovery and decision-making. To maintain a healthy ecosystem, vetting systems filter out low-quality apps with vulnerabilities or

compatibility issues. Ensuring consistency between apps and their metadata is crucial, as inconsistencies can negatively affect user experience. Such dissatisfaction may extend to the overall perception of the app store itself, emphasizing the need for maintaining alignment between app functionality and its metadata.
⟶**Representative Works:** Gorla et al. [39] have proposed to check app behaviour against app descriptions as they believe that there is no guarantee the code of the app does what it claims to do when uploaded to the app store. Their experimental results on a set of 22,500+ Android apps show that such inconsistency indeed exists in the community, confirming the hypothesis that the app store does not yet perform consistency checks at the time when apps are uploaded.

(25) **[App Store] Compliance Check.** Except for consistency checks, there is also a need to perform compliance checks before allowing mobile apps submitted to app stores. There are various policies that mobile apps need to follow. Such policies include the ones made by the government (e.g., the General Data Protection Regulation (GDPR) by the European Union), by the app store itself (e.g., the Spam and Minimum Functionality policies by Google Play), as well as the ones made by certain libraries (the content policies and behavioural policies by AdMob & AdSense.) These compliance checks should be also conducted for vetting OpenHarmony apps and hence dedicated efforts are needed to implement such approaches.
⟶**Representative Works:** Fan et al. [30] have conducted a study to explore the violations of GDPR compliance in Android eHealth apps. Their experimental study shows that such violations (including the incompleteness of privacy policy, the inconsistency of data collection, and the insecurity of data transmission) are indeed widely presented in the Android community.

(26) **[App Review] Human Values.** Mobile apps are developed for users, making it essential to align with human values. Violations of values like privacy, fairness, integrity, curiosity, honesty, or social justice can cause severe negative impacts. Early identification of such violations allows developers to address and mitigate them before release. Similarly, OpenHarmony should prioritize human values and support violation detection methods.
⟶**Representative Works:** Obie et al. [77] have presented to the MSE community the first study about human values-violation in app reviews given by real-world app users. Through 22,119 app reviews collected from the Google Play store, the authors find that 26.5% of the reviews contained text indicating user-perceived violations of human values, with benevolence and self-direction as the most violated value categories.

(27) **[Other] Black Market Analysis.** The rapid growth of the mobile ecosystem has attracted attackers seeking illegal profits, such as injecting ads into benign apps, sending SMS to premium-rate numbers, or collecting and selling user data for malicious purposes. Researchers term these activities the *black market* and have worked to understand and counter them. Similar risks exist for OpenHarmony, necessitating efforts to mitigate the black market, inviting researchers to explore this critical area collaboratively.
⟶**Representative Works:** Gao et al. [35] have conducted an exploratory study to demystify illegal mobile gambling apps, which have become one of the most popular and lucrative underground businesses. Their study reveals that, in order to bypass the strict regulations from both government authorities and app markets, the devious app authors have developed a number of covert channels to distribute their apps and abused fourth-party payment services to gain profits.

4.4 New Research Opportunities

- **LLM-based SE Approaches for OpenHarmony.** As summarized in Section 2, the majority of Mobile Software Engineering research works focus on the analyzing phase. There are only a limited number of studies focusing on app development phases. This does make sense as Android app development has already been quite mature (with a lot of support from Google and the community) when our fellow researchers

jumped into this field. This is, however, not the case for OpenHarmony. Indeed, OpenHarmony is still at a very early stage, with only a small number of apps developed and a limited number of third-party libraries made available to the community. It will be extremely beneficial to the OpenHarmony community if there are more works proposed to facilitate the development of OpenHarmony apps. Now, with the fast development of large language models (especially the development-focused ones such as Github's Copilot), we feel this is an even better opportunity to support that now. LLMs could help developers quickly learn the basic knowledge of OpenHarmony, understand the usage of APIs, automatically generate code (one line or multiple lines), generate unit test cases, recommend repair options, etc.

- **Cross-platfrom Framework for Supporting OpenHarmony.** To embrace the idea of developing once, running everywhere, the MSE community has invented the so-called cross-platform frameworks such as ReactNative and Flutter to support that. These cross-platform frameworks by themselves have defined a way to develop the universal app. For example, with ReactNative, the codebase of the app is usually formed via Javascript. This codebase can then be compiled into both a native Android app and a native iOS app. The best part of using cross-language platforms is that the app's maintenance is also unified. No matter it is to fix bugs or add new features, it only needs to be done once. Considering this great benefit, we believe it will be extremely helpful to OpenHarmony's ecosystem if these cross-platform frameworks can support OpenHarmony. In that case, all the existing apps that are developed via cross-platform frameworks can be directly running on OpenHarmony devices. Therefore, we highly recommend our fellow researchers considering exploring this research direction.
- **Learn from Android/iOS.** In this work, we have summarized lots of Android/iOS-related approaches and believe it is necessary to learn from them by building dedicated approaches for OpenHarmony. While that is certainly true, we also feel that there is a need to learn from the large number of artifacts accumulated in Android and iOS. Indeed, the MSE community has gained a lot of artifacts, including millions of real-world apps, thousands of open-source apps, documentation, question-and-answer records, user reviews, etc. Although harvested from different platforms, we argue that these artifacts could be still useful for supporting the implementation of OpenHarmony-related tasks. For example, one possibility is to explore the direction of automatically transforming the Java-written Android apps (or Swift-written iOS apps) to ArkTS-written OpenHarmony apps. In this work, we also invite our fellow researchers to explore this direction, flourishing the OpenHarmony ecosystem by standing on the shoulders of giants.

5 Discussion

OpenHarmony, as an emerging mobile platform, is still in its early stage, and so is OpenHarmony-focused software engineering research. As summarized previously, although there are plenty of opportunities for our fellow researchers to explore in this field, there are still various challenges that need to be addressed. In this section, we highlight some of the representative ones.

5.1 Challenges in App/Library Development.

In this work, we have highlighted the gaps that require to be filled in order to catch up with the popular mobile platforms (i.e., Android and iOS). Towards filling the gaps we argue that there are still a number of challenges that need to be addressed.

Lacking Data for (AI-based) Learning. The rise of large language models has been demonstrated to be promising for automated code generation, automated test case generation, library API recommendation, etc. However, it is not yet possible to directly achieve that for OpenHarmony as there is generally no data available for training (or fine-tuning). Even with a set of OpenHarmony-related software data (e.g., ArkTS code and its comments), there is also a requirement to further distil high-quality ones in order to achieve a highly precise

large language model, as the performance of large language models is known to be highly correlated with the quality of the training dataset.

Lacking Third-party Libraries. At the moment, there are only a limited number of libraries (in ArkTS) available for supporting the implementation of OpenHarmony apps. The lack of third-party libraries makes it difficult for developers to implement OpenHarmony apps as many of the functions need to be developed from scratch. To fill this gap, the OpenHarmony community is currently encouraging practitioners and researchers to translate popular libraries in other languages to ArkTS. However, this simple translation campaign will introduce another challenge, which is to keep updating the library following the updates of the original version. To that end, we argue that dedicated efforts are required to ensure the maintainability of these libraries.

5.2 Challenges in App/Library Analysis

After app (or library) development, there is a strong need to ensure that the app/library satisfies the requirements and is of high quality. The relevant challenges include the newly designed system architecture of OpenHarmony, the comprehensive GUI interactions, the newly introduced app programming language, etc. We now summarize the representative ones.

System-related Challenges. The Android system has introduced various challenges to the software engineering community in order to develop automated approaches to analyze Android apps. First, Android takes components to construct apps, for which the components themselves are independently developed. The components will not be directly connected at the code side and the actual invocation (via the so-called Inter-Component Communication (ICC) mechanism) will be done over the system. This ICC mechanism could also be leveraged to implement inter-app communications, making it a challenge to perform inter-app analyses. Second, the components in Android are designed to be run over a set of pre-defined methods (known as lifecycle methods) that will be triggered by the system following a certain order. These lifecycle methods are not connected at the code site as well, making it also a challenge for static app analysis (from the analyzer's point of view, there is no relationship between two lifecycle methods, despite they may be continuously called by the system). Third, similar to that of lifecycle methods, there are callback methods that are not directly connected to the app code as well. These callback methods are directly invoked by the system when certain events (either system events such as receiving an SMS or UI events such as clicking a button) are triggered. OpenHarmony generally shares the same challenges as that of Android.

GUI-related Challenges. The GUI part has been known to be a challenge for precisely analyzing Android apps. First of all, a given GUI page often contains a comprehensive view tree that includes various widgets with different types positioned via different layout strategies. The widgets in the GUI page are further associated with interactive actions (e.g., a button is associated with a click event). Furthermore, a given GUI page may contain different groups of widgets that will only be rendered if a certain condition is satisfied. In OpenHarmony, the analysis of GUI pages is even more challenging as its design principle encourages to use of a single component (i.e., Ability) to implement multiple visual pages, which would be implemented via multiple components (i.e., Activities, one page per Activity) in Android.

Language-induced Challenges. The language used to implement mobile apps per se may introduce challenges to the software engineering community. For example, in the Android world, the reflection mechanism (inherited from Java) has been known to be a challenge for static analysis. OpenHarmony takes a new language called ArkTS for developers to implement OpenHarmony apps and the ArkTS language per se may introduce various challenges to the software engineering community as well. Indeed, ArkTS allows defining functions with optional parameters and default parameters, which may cause inconsistency between the function signature and its usage in practice.

6 Related Work

OpenHarmony software engineering is in its early stage and there are only limited works contributed to this field. Indeed, as highlighted in Section 3.3, there are only 8 papers presented on this aspect. In this section, we will not discuss these OpenHarmony-related works anymore. Instead, we take this opportunity to highlight related works that provide a research roadmap or position statement for guiding a new research field, or a survey including literature reviews for summarizing a mature research direction. We now highlight the representative ones.

Research Roadmap. One of the most representative research roadmap reports is the one presented by Cheng et al. [23] who have proposed ato conduct software engineering research roadmap for self-adaptive systems, following. After thorough discussions among the authors at a Dagstuhl seminar. They identified four on Software Engineering for Self-Adaptive Systems, the authors have identified four views that are deemed essential views forto the software engineering in this domain. of self-adaptive systems. For each view, the authors then summarize the state-of-the-art and highlight the challenges to enable software to autonomously handlehat should be addressed in order to achieve the final goal, i.e., the software is able to automatically cope with the complexity of mtodernay's software-intensive systems. The authors released another version (called the second research roadmap) five years later after the success of the first version. The goal of this second roadmap paper [24] remains the same, i.e., to summarize the state-of-the-art and to identify critical challenges for the systematic software engineering of self-adaptive systems. Other representative research roadmap papers include the one proposed by France et al. [31] who advocate model-driven development of complex software. Both of these works have summarized the state-of-the-art and challenges faced by ongoing research activities. More recently, McDermott et al. [74] present a research roadmap about Artificial Intelligence for Software Engineering (AI4SE) and Software Engineering for Artificial Intelligence (SE4AI), presenting key aspects aiming at enabling traditional systems engineering practice automation (AI4SE), and encourage new systems engineering practices supporting a new wave of automated, adaptive, and learning systems (SE4AI).

Literature Review. A literature review surveys scholarly sources on a specific topic, offering an overview of the state-of-the-art supported by a critical evaluation. Beyond reflecting on past research, it provides a clear understanding of current knowledge, guiding future research directions. Due to these benefits, this work focuses on surveying literature review papers rather than primary publications in mobile software engineering. Notably, conducting a survey of surveys is not new to the community. Our fellow researchers have explored this type of study in various domains when the number of primary publications kept increasing until it became difficult to follow the growing body of literature papers in the field. For example, AI-Zewairi et al. [3] have conducted a survey of surveys related to agile software development methodologies, which have gained rigorous attention in the software engineering community with an excessive number of research studies published.

7 Conclusion

It has been evidenced that summarizing the research roadmap for a given topic is important as it highlights various research opportunities that communicate broad research goals to the community, connects researchers working on individual projects to larger impact opportunities, and helps professional societies and practitioners focus on more strategic goals. Following this guidance, in this work, we propose to the community a research roadmap about software engineering for OpenHarmony, aiming at creating a synergy for the various stakeholders to work together to make OpenHarmony a successful mobile platform. Specifically, we have summarized the status quo of OpenHarmony software engineering research, for which we show OpenHarmony research is still in its early stage. We then highlight the research opportunities by summarizing the gap between OpenHarmony research and Mobile software engineering research, which is summarized through a survey of literature review papers. After that, we briefly discuss the challenges in order to fill such a gap.

Acknowledgements

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments that have led to important improvements in several parts of the manuscript. This work is supported by National Natural Science Foundation of China under Grant Nos (62141209, 61932007). Grundy is supported by ARC Laureate Fellowship FL190100035.

References

[1] Naveed Ahmad, Aimal Rextin, and Um E Kulsoom. 2018. Perspectives on usability guidelines for smartphone applications: An empirical investigation and systematic literature review. *IST* 94 (2018), 130–149.
[2] Afnan A. Al-Subaihin, Federica Sarro, Sue Black, Licia Capra, and Mark Harman. 2021. App Store Effects on Software Engineering Practices. *TSE* 47, 2 (2021), 300–319.
[3] Malek Al-Zewairi et al. 2017. Agile software development methodologies: Survey of surveys. *Journal of Computer and Communications* 5, 05 (2017), 74.
[4] Mughees Ali, Saif Ur Rehman Khan, and Shahid Hussain. 2021. Self-adaptation in smartphone applications: Current state-of-the-art techniques, challenges, and future directions. *Data & Knowledge Engineering* 136 (2021), 101929.
[5] Abdullah Altaleb and Andrew Gravell. 2018. Effort estimation across Mobile app platforms using agile processes: a systematic literature review. *Journal of Software* 13, 4 (2018), 242.
[6] Domenico Amalfitano et al. 2012. Using GUI ripping for automated testing of Android applications. In *ASE*. 258–261.
[7] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* 11 (2020), 100403.
[8] Steven Arzt and others. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.
[9] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *CCS*. 217–228.
[10] Autili and others. 2021. Software engineering techniques for statically analyzing mobile apps: research trends, characteristics, and potential for industrial adoption. *Journal of Internet Services and Applications* 12 (2021), 1–60.
[11] Muneera Bano, Didar Zowghi, and Naveed Ikram. 2014. Systematic reviews in requirements engineering: A tertiary study. In *EmpiRE*. IEEE, 9–16.
[12] Konstantia Barmpatsalou, Tiago Cruz, Edmundo Monteiro, and Paulo Simoes. 2018. Current and future trends in mobile device forensics: A survey. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–31.
[13] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2014. Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android. *TSE* (2014).
[14] Alexandre Bartel and others. 2012. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *SOAP*. 27–38.
[15] Alastair R Beresford et al. 2011. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*. 49–54.
[16] Andreas Biørn-Hansen, Tor-Morten Grønli, and Gheorghita Ghinea. 2018. A survey and taxonomy of core concepts and research challenges in cross-platform mobile development. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–34.
[17] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from applying the systematic literature review process within the software engineering domain. *JSS* 80, 4 (2007), 571–583.
[18] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. 2021. Vins: Visual search for mobile user interface design. In *CHI*. 1–14.
[19] Marimuthu C., K. Chandrasekaran, and Sridhar Chimalakonda. 2020. Energy Diagnosis of Android Applications: A Thematic Taxonomy and Survey. *ACM Comput. Surv.* 53, 6, Article 117 (dec 2020), 36 pages.
[20] Tinggui Chen, Chu Zhang, Jianjun Yang, and Guodong Cong. 2022. Grounded Theory-Based User Needs Mining and Its Impact on APP Downloads: Exampled With WeChat APP. *Frontiers in Psychology* 13 (2022), 875310.
[21] Valerio Costamagna and Cong Zheng. 2016. Artdroid: A virtual-method hooking framework on android art runtime.. In *IMPS@ ESSoS*. 20–28.
[22] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. Andarwin: Scalable detection of semantically similar android applications. In *ESORICS*. Springer, 182–199.
[23] Rogério De Lemos et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32.
[24] Rogério De Lemos et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32.

[25] Omar De Munk and Ivano Malavolta. 2021. Measurement-based experiments on the mobile web: A systematic mapping study. *EASE* (2021), 191–200.
[26] Paula Delgado-Santos, Giuseppe Stragapede, Ruben Tolosana, Richard Guest, Farzin Deravi, and Ruben Vera-Rodriguez. 2022. A Survey of Privacy Vulnerabilities of Mobile Device Sensors. *ACM Comput. Surv.* 54, 11s, Article 224 (sep 2022), 30 pages.
[27] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *SecureComm*. Springer, 172–192.
[28] Fahimeh Ebrahimi, Miroslav Tushev, and Anas Mahmoud. 2021. Mobile app privacy in software engineering research: A systematic mapping study. *IST* 133 (2021), 106466.
[29] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *TOCS* 32, 2 (2014), 1–29.
[30] Ming Fan et al. 2020. An Empirical Evaluation of GDPR Compliance Violations in Android mHealth Apps. In *ISSRE*. 253–264.
[31] Robert France and Bernhard Rumpe. 2007. Model-driven development of complex software: A research roadmap. In *FOSE*. IEEE, 37–54.
[32] Rita Francese et al. 2017. Mobile app development and management: results from a qualitative investigation. In *MOBILESoft*. IEEE, 133–143.
[33] Jun Gao et al. 2019. On the Evolution of Mobile App Complexity. In *ICECCS 2019*.
[34] Jun Gao et al. 2019. Understanding the Evolution of Android App Vulnerabilities. *TRel* (2019).
[35] Yuhao Gao et al. 2021. Demystifying Illegal Mobile Gambling Apps. In *WWW 2021*.
[36] Xiuting Ge, Shengcheng Yu, Chunrong Fang, Qi Zhu, and Zhihong Zhao. 2022. Leveraging android automated testing to assist crowdsourced testing. *TSE* 49, 4 (2022), 2318–2336.
[37] Necmiye Genc-Nayebi and Alain Abran. 2017. A systematic literature review: Opinion mining studies from mobile app store user reviews. *JSS* 125 (2017), 207–219.
[38] Lorenzo Gomez et al. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *ICSE*. IEEE, 72–81.
[39] Alessandra Gorla et al. 2014. Checking App Behavior against App Descriptions *(ICSE 2014)*. 1025–1035.
[40] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. 2021. A survey of performance optimization for mobile applications. *TSE* 48, 8 (2021), 2879–2904.
[41] Seyed Amir Hoseini-Tabatabaei, Alexander Gluhak, and Rahim Tafazolli. 2013. A survey on smartphone-based systems for opportunistic user context recognition. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 1–51.
[42] Ronald Jabangwe, Henry Edison, and Anh Nguyen Duc. 2018. Software engineering process models for mobile app development: A systematic literature review. *JSS* 145 (2018), 98–111.
[43] Mona Erfani Joorabchi et al. 2013. Real challenges in mobile app development. In *ESEM*. IEEE, 15–24.
[44] Misael C. Júnior, Domenico Amalfitano, Lina Garcés, Anna Rita Fasolino, Stevão A. Andrade, and Márcio Delamaro. 2022. Dynamic Testing Techniques of Non-Functional Requirements in Mobile Apps: A Systematic Mapping Study. *ACM Comput. Surv.* 54, 10s, Article 214 (sep 2022), 38 pages.
[45] Anureet Kaur and Kulwant Kaur. 2019. Investigation on test effort estimation of mobile applications: Systematic literature review and survey. *Information and Software technology* 110 (2019), 56–77.
[46] Staffs Keele et al. 2007. Guidelines for performing systematic literature reviews in software engineering. *Technical report* (2007).
[47] Seoyeon Kim, Jisu Park, Jinman Jung, Seongbae Eun, Y-S Yun, S So, B Kim, H Min, and J Heo. 2018. Identifying UI widgets of mobile applications from sketch images. (2018).
[48] Young Geun Kim, Joonho Kong, and Sung Woo Chung. 2018. A Survey on Recent OS-Level Energy Management Techniques for Mobile Processing Units. *TPDS* 29, 10 (2018), 2388–2401.
[49] Barbara Kitchenham et al. 2010. Systematic literature reviews in software engineering–a tertiary study. *Information and software technology* 52, 8 (2010), 792–805.
[50] Barbara Kitchenham and Pearl Brereton. 2013. A systematic review of systematic review process research in software engineering. *IST* 55, 12 (2013), 2049–2075.
[51] Pingfan Kong et al. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).
[52] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* 68, 1 (2019), 45–66.
[53] Zoe Kotti, Rafaila Galanopoulou, and Diomidis Spinellis. 2023. Machine learning for software engineering: A tertiary study. *Comput. Surveys* 55, 12 (2023), 1–39.
[54] Patrick Lam et al. 2011. The Soot framework for Java program analysis: a retrospective. In *CETUS*, Vol. 15.
[55] Hansoo Lee, Joonyoung Park, and Uichin Lee. 2022. A Systematic Survey on Android API Usage for Data-Driven Analytics with Smartphones. *ACM Comput. Surv.* 55, 5, Article 104 (dec 2022), 38 pages.
[56] Li Li, , et al. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* (2017).
[57] Li Li et al. 2020. CDA: Characterising Deprecated Android APIs. *EMSE* (2020).

[58] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *TSE* 47, 4 (2019), 676–693.
[59] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *ICSME*.
[60] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. 2017. AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. *arXiv preprint arXiv:1709.05281* (2017).
[61] Shu-Hui Li, Jia-Jyun Hsu, Chih-Ya Chang, Pin-Hsuan Chen, and Neng-Hao Yu. 2017. Xketch: A sketch-based prototyping tool to accelerate mobile app design process. In *DIS*. 301–304.
[62] Pei Liu, Mattia Fazzini, John Grundy, and Li Li. 2022. Do Customized Android Frameworks Keep Pace with Android?. In *MSR*.
[63] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. 2021. Identifying and Characterizing Silently-Evolved Methods in the Android API. In *ICSE-SEIP*.
[64] Pei Liu, Li Li, Yanjie Zhao, Xiaoyu Sun, and John Grundy. 2020. AndroZooOpen: Collecting Large-scale Open Source Android Apps for the Research Community. In *MSR-Data*.
[65] PEI LIU, YANJIE ZHAO, MATTIA FAZZINI, HAIPENG CAI, JOHN GRUNDY, and LI LI. 2023. Automatically Detecting Incompatible Android APIs. *TSE* (2023).
[66] Yue Liu et al. 2022. Deep Learning for Android Malware Defenses: a Systematic Literature Review. *ACM Computing Surveys (CSUR)* (2022).
[67] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep Learning for Android Malware Defenses: A Systematic Literature Review. *ACM Comput. Surv.* 55, 8, Article 153 (dec 2022), 36 pages.
[68] Zhe Liu et al. 2023. Ex pede Herculem: Augmenting Activity Transition Graph for Apps via Graph Convolution Network. In *ICSE*. IEEE, 1983–1995.
[69] Chu Luo, Jorge Goncalves, Eduardo Velloso, and Vassilis Kostakos. 2020. A Survey of Context Simulation for Testing Mobile Context-Aware Applications. *ACM Comput. Surv.* 53, 1, Article 21 (feb 2020), 39 pages.
[70] Lannan Luo et al. 2019. Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation. *IEEE Transactions on Mobile Computing* 19, 12 (2019), 2946–2964.
[71] Pascal Maniriho et al. 2024. A survey of recent advances in deep learning models for detecting malware in desktop and mobile platforms. *Comput. Surveys* 56, 6 (2024), 1–41.
[72] Alexandru Marginean et al. 2019. Sapfix: Automated end-to-end repair at scale. In *ICSE-SEIP*. IEEE, 269–278.
[73] William Martin et al. 2016. A survey of app store analysis for software engineering. *TSE* 43, 9 (2016), 817–847.
[74] Tom McDermott et al. 2020. AI4SE and SE4AI: A research roadmap. *Insight* 23, 1 (2020), 8–14.
[75] Walter T Nakamura et al. 2022. What factors affect the UX in mobile apps? A systematic mapping study on the analysis of app store reviews. *JSS* 193 (2022), 111462.
[76] Liming Nie, Kabir Sulaiman Said, Lingfei Ma, Yaowen Zheng, and Yangyang Zhao. 2023. A systematic mapping study for graphical user interface testing on mobile apps. *IET Software* 17, 3 (2023), 249–267.
[77] Humphrey Obie, Waqar Hussain, Xin Xia, John Grundy, Li Li, Burak Turhan, Jon Whittle, and Mojtaba Shahin. 2021. A First Look at Human Values-Violation in App Reviews. In *ICSE-SEIS*.
[78] Humphrey Obie, Idowu Ilekura, Hung Du, Mojtaba Shahin, John Grundy, Li Li, Jon Whittle, and Burak Turhan. 2022. On the Violation of Honesty in Mobile Apps: Automated Detection and Categories. In *MSR*.
[79] openharmony. 2024. 47 new products qualified for OpenHarmony compatibility test. https://www.huaweicentral.com/47-new-products-qualified-for-openharmony-compatibility-test-in-june-2024/.
[80] OpenHarmony. 2024. ArkAnalyzer: The static analysis framework for OpenHarmony. https://gitee.com/openharmony-sig/arkanalyzer.
[81] OpenHarmony. 2024. OpenHarmony/ostest_wukong. https://gitee.com/openharmony/ostest_wukong. Accessed: 2024-10-22.
[82] openharmony. 2024. Security Issue Response Team Work Charter. https://gitee.com/openharmony/security/blob/master/README_en.md.
[83] OpenHarmony. 2024. UiTest Features. https://gitee.com/openharmony/testfwk_arkxtest.
[84] Fabio Palomba et al. 2018. Crowdsourcing user reviews to support the evolution of mobile apps. *JSS* 137 (2018), 143–162.
[85] Orlando RE Pereira and Joel JPC Rodrigues. 2013. Survey and analysis of current mobile learning applications and technologies. *ACM Computing Surveys (CSUR)* 46, 2 (2013), 1–35.
[86] Jorge Pérez, Jessica Díaz, Javier Garcia-Martin, and Bernardo Tabuenca. 2020. Systematic literature reviews in software engineering—Enhancement of the study selection process using Cohen's kappa statistic. *JSS* 168 (2020), 110657.
[87] Fangze Qiu, Huaxiao Huang, and Yuji Dong. 2022. A Re-configurable Interaction Model in Distributed IoT Environment. In *CyberC*. IEEE, 80–86.
[88] Guoying Qiu et al. 2023. Differentiated Location Privacy Protection in Mobile Communication Services: A Survey from the Semantic Perception Perspective. *Comput. Surveys* 56, 3 (2023), 1–36.

[89] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. 2020. A Survey of Android Malware Detection with Deep Neural Models. *ACM Comput. Surv.* 53, 6, Article 126 (dec 2020), 36 pages.
[90] Siegfried Rasthofer and Aothers. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. *NDSS* (2014).
[91] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *ICSE*. IEEE, 300–311.
[92] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. 2013. Firedroid: Hardening security in almost-stock android. In *ACSAC*. 319–328.
[93] Alireza Sadeghi et al. 2016. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *TSE* 43, 6 (2016), 492–530.
[94] Jordan Samhi et al. 2022. Jucify: A step towards android code unification for enhanced static analysis. In *ICSE*. 1232–1244.
[95] Joanna Cecilia Da Silva Santos and Julian Dolby. 2022. Program Analysis using WALA. In *FSE*.
[96] Janaka Senanayake, Harsha Kalutarage, Mhd Omar Al-Kadri, Andrei Petrovski, and Luca Piras. 2023. Android Source Code Vulnerability Detection: A Systematic Literature Review. *CSUR* 55, 9, Article 187 (jan 2023), 37 pages.
[97] Basit Shahzad, Abdullatif M Abdullatif, Kashif Saleem, and Wasif Jameel. 2017. Socio-technical challenges and mitigation guidelines in developing mobile healthcare applications. *JMIHI* 7, 3 (2017), 704–712.
[98] Md Shamsujjoha, John Grundy, Li Li, Hourieh Khalajzadeh, and Qinghua Lu. 2021. Developing mobile applications via model driven development: a systematic literature review. *IST* 140 (2021), 106693.
[99] Camila Silva et al. 2018. A survey on the tool support for the automatic evaluation of mobile accessibility. In *DSAI*. 286–293.
[100] Henrique Neves Silva, Jackson Prado Lima, Silvia Regina Vergilio, and Andre Takeshi Endo. 2022. A mapping study on mutation testing for mobile applications. *Software Testing, Verification and Reliability* 32, 8 (2022), e1801.
[101] Michael Spreitzenbarth et al. 2013. Mobile-sandbox: having a deeper look into android applications. In *SAC*. 1808–1815.
[102] Ting Su et al. 2017. Guided, stochastic model-based GUI testing of Android apps. In *FSE*. 245–256.
[103] Sufatrio, Darell JJ Tan, Tong-Wei Chua, and Vrizlynn LL Thing. 2015. Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 1–45.
[104] Xiaoyu Sun, Xiao Chen, Yonghui Liu, John Grundy, and year=2023 publisher=IEEE Li, journal=TSE. [n. d.]. Taming Android Fragmentation through Lightweight Crowdsourced Testing. ([n. d.]).
[105] Porfirio Tramontana et al. 2019. Automated functional testing of mobile applications: a systematic mapping study. *SQJ* 27 (2019), 149–201.
[106] Hai Vu-Ngoc et al. 2018. Quality of flow diagram in systematic review and/or meta-analysis. *PloS one* 13, 6 (2018), e0195955.
[107] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. 2018. Automated hybrid analysis of android malware through augmenting fuzzing with forced execution. *IEEE Transactions on Mobile Computing* 18, 12 (2018), 2768–2782.
[108] Yihui Wang, Huaxiao Liu, Shanquan Gao, and Xiao Tang. 2023. Animation2API: API Recommendation for the Implementation of Android UI Animations. *TSE* (2023).
[109] Ying Wang, Yibo Wang, Sinan Wang, Yepang Liu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2022. Runtime permission issues in android apps: Taxonomy, practices, and ways forward. *TSE* 49, 1 (2022), 185–210.
[110] Fengguo Wei et al. 2018. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *CCS*. 1137–1150.
[111] Chathrie Wimalasooriya, Sherlock A Licorish, Daniel Alencar da Costa, and Stephen G MacDonell. 2022. A systematic mapping study addressing the reliability of mobile applications: The need to move beyond testing reliability. *JSS* 186 (2022), 111166.
[112] Zhiqiang Wu, Xin Chen, and Scott Uk-Jin Lee. 2023. A systematic literature review on Android-specific smells. *JSS* 201 (2023), 111677.
[113] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, et al. 2016. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–47.
[114] Lei Xue et al. 2018. NDroid: Toward tracking information flows across multiple Android contexts. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 814–828.
[115] Lei Xue et al. 2021. Happer: Unpacking android apps via a hardware-assisted approach. In *SP*. IEEE, 1641–1658.
[116] Lei Xue et al. 2021. Parema: an unpacking framework for demystifying VM-based Android packers. In *ISSTA*. 152–164.
[117] Lei Xue and Zothers. 2020. Packergrind: An adaptive unpacking system for android apps. *TSE* 48, 2 (2020), 551–570.
[118] Ping Yan and Zheng Yan. 2018. A survey on dynamic mobile malware detection. *SQJ* 26, 3 (2018), 891–919.
[119] Yuri D. V. Yasuda, Luiz Eduardo G. Martins, and Fabio A. M. Cappabianco. 2020. Autonomous Visual Navigation for Mobile Robots: A Systematic Literature Review. *ACM Comput. Surv.* 53, 1, Article 13 (feb 2020), 34 pages.
[120] Samer Zein et al. 2016. A systematic mapping study of mobile application testing techniques. *JSS* 117 (2016), 334–356.
[121] Samer Zein, Norsaremah Salleh, and John Grundy. 2016. A systematic mapping study of mobile application testing techniques. *J. Syst. Softw.* 117 (2016), 334–356.

[122] Samer Zein, Norsaremah Salleh, and John Grundy. 2023. Systematic reviews in mobile app software engineering: A tertiary study. *IST* 164 (2023), 107323.
[123] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. 2021. Research on third-party libraries in android apps: A taxonomy and systematic literature review. *TSE* (2021).
[124] Xian Zhan, Tao Zhang, and Yutian Tang. 2019. A comparative study of android repackaged apps detection techniques. In *SANER*. IEEE, 321–331.
[125] Xiangyu Zhang et al. 2023. Scene-Driven Exploration and GUI Modeling for Android Apps. *arXiv* (2023).
[126] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. Dexhunter: toward extracting hidden code from packed android applications. In *ESORICS*. Springer, 293–311.
[127] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards Automatically Repairing Compatibility Issues in Published Android Apps. In *ICSE*.
[128] Yanjie Zhao, Li Li, Xiaoyu Sun, Pei Liu, and John Grundy. 2021. Icon2Code: Recommending code implementations for Android GUI components. *IST* 138 (2021), 106619.
[129] Yanjie Zhao, Li Li, Haoyu Wang, Qiang He, and John Grundy. 2022. APIMatchmaker: Matching the Right APIs for Supporting the Development of Android Apps. *TSE* (2022).
[130] Hao Zhou et al. 2021. Finding the missing piece: permission specification analysis for Android NDK. In *ASE*. IEEE, 505–516.
[131] Hao Zhou et al. 2022. NCScope: hardware-assisted analyzer for native code in Android apps. In *ISSTA*. 629–641.
[132] Hao Zhou et al. 2022. Uncovering Intent based Leak of Sensitive Data in Android Framework. In *CCS*. 3239–3252.

Received 24 January 2024; revised 23 January 2025; accepted 3 February 2025