The Journal of Systems & Software 201 (2023) 111677

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A systematic literature review on Android-specific smells ☆

Zhiqiang Wu [1], Xin Chen [1], Scott Uk-Jin Lee *

Department of Computer Science & Engineering, Hanyang University, South Korea

ARTICLE INFO

Article history:
Received 9 June 2022
Received in revised form 27 December 2022
Accepted 9 March 2023
Available online 13 March 2023

Keywords:
Android
Code smell
Systematic literature review

ABSTRACT

Context: Code smells are well-known concepts in Object-Oriented (OO) programs as symptoms that negatively impact software quality and cause long-term issues. However, the domain-specific smells in Android have not yet been investigated well. Android smells often refer to the misuse of mobile SDK and causes of performance, accessibility, and efficiency issues that end-users can perceive.
Objective: This study aims to provide a clear overview of state-of-the-art techniques for addressing Android-specific code smells to understand existing methods and open challenges, which help the community understand the significance of Android smells and the current status of research.
Methods: We conducted a Systematic Literature Review of 4,820 distinct papers published until 2021, following a consolidated methodology applied in software engineering. 35 primary studies were selected.
Results: The known Android smells cannot be treated equally in the proposed approaches, as they mainly focus on detecting performance-related smells. The proposed approaches capture various features to detect smell instances using different analysis techniques in Android applications. In addition, the Android community continuously identifies new types of smells to improve apps' quality.
Conclusion: The research community still encounters several challenges. Thus, this paper outlines various directions for the necessary investigation as future work.

1. Introduction

Software systems need to be continuously updated to cope with new requirements and long-term maintenance. High-quality source code plays a critical role in this context since it should be easy to understand, analyze, maintain, and reuse (Jiang et al., 2014). However, practitioners eventually produce sub-optimal solutions during development owing to various factors, such as time pressure and limited resources (Kuutila et al., 2021). That is, such sub-optimal solutions are bad practices in software development that violate the fundamental software design principle (i.e., SOLID Martin et al., 2018), degrading internal software quality (Martins et al., 2021). These situations are well-known as code smells in Object-Oriented (OO) programs (De Stefano et al., 2020). Although code smells may hinder the evolution of software in practice, they may introduce some short-term benefits. For instance, developers may have insufficient time to implement functionalities with thoughtful consideration before the deadline. To deliver software on time, code smells are easily introduced as sub-optimal solutions, which provide higher productivity. According to an empirical study, 27% of maintenance issues are caused by code smells (Yamashita and Moonen, 2013), which have become a backlog debt during development. Based on the aforementioned issues, the software is difficult to maintain since code smells degrade the maintainability and understandability of the software. Developers eventually have to make additional efforts to understand the structure of software and refactor code smells before implementing new functionalities.

In recent years, mobile applications (apps) have dominated the major software market (Anon, 2022b), particularly Android. Since mobile apps rely on the same software development basics as traditional desktop software (e.g., OO programming languages and infrastructures), they also retain the possibility of introducing OO smells in the source code. In addition, current mobile apps integrate various powerful features into our daily lives. If these features cannot be implemented with the best practices, mobile devices may cost hardware resources to handle them extensively, which causes battery drain and performance decline. Especially, Android, an open-source system, runs on various environments with different hardware specs. More specifically, one Android app installs on different mobile devices, showing discrepancies in performance, efficiency, and user experiences due to the capacities of different hardware. Compared to iOS, Android practitioners should pay more attention to developing apps with best practices

☆ Editor: Shane McIntosh.
* Corresponding author.
E-mail addresses: wzq0515@hanyang.ac.kr (Z. Wu), xxtx0122@hanyang.ac.kr (X. Chen), scottlee@hanyang.ac.kr (S.U.-J. Lee).
[1] These authors contributed equally to this work.

https://doi.org/10.1016/j.jss.2023.111677
0164-1212/

for adapting various hardware specs. Therefore, considering the limitations of mobile devices such as memory, CPU, energy, and a peculiar framework is necessary (Hecht et al., 2015a). In particular, Reimann (2014) proposed a catalog of Android-specific code smells that violate best practices in Android development. Android code smells differ from OO code smells because they often refer to misuse of the platform SDK, which leads to various issues regarding performance, memory and energy efficiency, and accessibility (Reimann and Aßmann, 2013).

To address these issues, the research community has investigated various aspects of Android-specific smells and proposed a wide range of program analyses to detect and repair Android smells (Palomba et al., 2017) compare the divergence of maintainability and quality between OO and Android smells (Peruma, 2019; Oliveira et al., 2018), uncover performance-related bad practices (Mazuera-Rozo et al., 2020), etc.. The studies have reported the negative impacts of Android-specific code smells on software quality (Habchi et al., 2021b), showing that Android smells are not harmful to functional requirements, but can hinder app performance and waste additional resources on memory, CPU, and energy (Palomba et al., 2019; Góis Mateus and Martinez, 2019; Lyu et al., 2019). For example, the code smell Durable WakeLock states that WakeLock cannot be released properly, increasing energy consumption. Android smells can be used as indicators of bad code practices, representing potential quality issues (Palomba et al., 2019).

In this paper, we aim to provide a clear overview of Android-specific smells, including body knowledge and open challenges, to remedy current lacks, helping the community understand the impacts of Android smells from multiple dimensions. This SLR provides various evidence for the research community to study android smells further to overcome the open challenges and develop more off-the-shelf tools for practitioners, facilitating that the domain of Android smells becomes mature. In addition, Android practitioners may be aware of the importance of Android-specific smells on performance, efficiency, and security issues. It encourages them to leverage the existing tools to eliminate Android smells from source code. To achieve this goal, we performed a Systematic Literature Review (SLR) between 2014 and 2021, considering the diversity of Android smells, negative impacts on quality, fundamental techniques to address smells, and evaluation methods. After thoroughly identifying the set of related publications, we conducted a trend analysis based on current open challenges and potential new research directions discovered. To the best of our knowledge, this is the first study to perform such a comprehensive analysis of Android smells.

The main contributions of this SLR are:

- A web repository[2] newly built to make our SLR results publicly available for reproducibility and extension.
- All known Android-specific smells from selected primary studies were enumerated to reveal convergent, divergent, and main findings, which helps the Android community understand the significance of Android smells.
- The key aspects of Android-specific smells were analyzed in detail to summarize the research efforts and results, which discloses the current shortages in this domain. It not only provides a clear research direction for the research community but also guides developers to leverage various off-the-shelf tools for eliminating Android smells.
- We disclosed a set of open challenges encountered by the current studies. It encourages researchers to build corresponding solutions for automatically detecting and refactoring Android smells without human intervention.

This SLR is structured as follows. Section 2 presents the background of code smells in the OO program and Android apps. Section 3 presents our research questions and the systematic literature review methodology. Section 4 presents the results of the research questions extracted from selected primary studies. In Section 5, we discuss the main findings of this SLR, and outline future challenges and research directions. Sections 6 and 7 discuss potential threats to the validity of this SLR and related work, respectively. Finally, the conclusions are presented in Section 8.

2. Background

This section provides preliminary details necessary for understanding the purposes, techniques, and key concerns of the reviewed research. Specifically, we introduce the concept of code smells in the OOP in Section 2.1 before investigating some domain details of Android-specific code smells and relevant research activities in Section 2.2.

2.1. Object-oriented code smells

In the 1990s, Flower et al. introduced *code smells* to describe symptoms observed in object-oriented programs that degrade the internal quality of software (Fowler et al., 1999). Unlike software bugs, code smells are not faults or crashes in software. These are a set of code patterns as indicators of sub-optimal implementation choices, which are usually caused by poor design and practices during implementation at the code level (Lacerda et al., 2020). In general, code smells can deteriorate the essential code quality aspects of software maintenance, such as understandability and changeability (i.e., non-functional attributes) (Boutaib et al., 2021). Such quality properties impair software evolution since end-users cannot perceive these quality properties at the code level (Sobrinho et al., 2021).

In the short term, code smells may introduce short-term benefits, even if their existence violates the fundamental design principles of software engineering (Sobrinho et al., 2021). However, in the long term, practitioners have to continuously make additional efforts to remedy such smells before coping with new requirements, also known as technical debt (Brown et al., 2010).

2.2. Android-specific code smells

2.2.1. Android smells

The traditional OO smells also occur in Android apps since they are written in OO programming languages (i.e., Java, Kotlin, and JavaScript). Moreover, Android apps possess exclusive characteristics for implementing various functionalities on mobile apps (i.e., the Android framework). Misusing the Android framework API also affects non-functional quality attributes, such as performance (Habchi, 2019). To overcome this domain issue, Reimann and Aßmann (2013) summarized a catalog[3] of 30 Android-specific code smells, which originated from the good and bad practices presented online in Android documentation. These dedicated smells impact various non-functional quality attributes in mobile apps, such as performance, memory, and energy efficiency.

Furthermore, Android apps run on mobile devices with limited hardware resources, such as memory, CPU, and screen size (Hecht et al., 2016). In other words, the common criteria of quality requirements in OO are not effective in Android development. Although many Android developers have worked on desktop software beforehand, they still adopt desktop practices without realizing their negative impacts on app performance (Habchi et al., 2019a). For instance, HashMap is a common data structure in Java. However, using a HashMap with a large data size is considered a code smell in Android, which is called Inefficient Data Structure (IDS).

[2] https://doi.org/10.6084/m9.figshare.20001788

[3] https://martinbrylski.github.io/android_smells

2.2.2. Studied activities

Several studies related to addressing Android smells in the domain of software engineering. The research community has proposed methods for detecting, refactoring, and prioritizing code smells.

Detection. The goal is to detect code smells from the source code. Researchers have proposed methods for smell detection that can be divided into five categories (Sharma and Spinellis, 2018). Google provides an official tool integrated into Android Studio for inspecting Android smells, named AndroidLint (Android, 2017). In addition, the strategies for detecting Android smells also include the below approaches.

- Metric-based. The most popular smell detection techniques employ metric-based methods. Generally, this method extracts a set of software metrics from source code in which code smells can be detected by capturing the characteristics of smells with suitable thresholds for metrics (Sharma and Spinellis, 2018). For instance, a god class can be detected using the following metrics: a class has high Access Of Foreign Data (AOFD), Weighted Method per Class (WMPC), and Tight Class Cohesion (TCC) (Salehie et al., 2006). To quantify what is "high", the research community conducted statistical methods and empirical studies to define project-specific thresholds (Maiga et al., 2012; Marinescu, 2004).
- Rule-based. Many smells cannot be detected using only existing software metrics, such as rebellious hierarchy, missing abstraction, and cyclic hierarchy (Suryanarayana et al., 2014). In such cases, to detect smells, rule-based methods combine software metrics and source code models, such as Abstract Syntax Tree (AST) (Gao et al., 2019), Control Flow Graph (Khan et al., 2021b), and Program Dependency Graph (Wang et al., 2017; Silva et al., 2020).
- Search-based. Each code smell impairs various quality attributes, reflecting different software metrics (Sharma and Spinellis, 2018). A search-based method leverages evolutionary algorithms for computed metrics, such as genetic algorithms (Shoenberger et al., 2017; Kessentini and Ouni, 2017), to generate the best detection rules that maximize the coverage of code smell based on single/multi-objectives.
- History-based. Palomba et al. (2014) proposed a new code smell detection technique based on source code evolution information. This technique mines historical commits from the version control system to detect divergent change smells that have frequently changed during past evolutions (Palomba et al., 2013) (i.e., change-proneness Khomh et al., 2009).
- Learning-based. A different line of work uses machine learning to detect code smells, thereby reinforcing search-based approaches (Lacerda et al., 2020). A typical learning-based method starts with a mathematical model representing code characteristics based on metrics and source-code models. Then, a chosen machine learning algorithm can be trained with a bulk of existing examples. Learning-based methods are excessively applied to automatically adjust the thresholds for cross-projects (Silva et al., 2020; Fang et al., 2020), capture the context of Android smells (Sharma et al., 2021), and analyze code semantics to measure code similarity (Gao et al., 2019; Zhang et al., 2019).

Refactoring. As defined by Opdyke (1992), refactoring is a pattern for re-organizing software components. It is the primary approach to remove smells by applying transformations that preserve the functional behaviors of the source code and improve the internal quality attributes of the software (Bavota et al., 2015). In general, refactoring a code smell involves two steps: *Smell localization*, in which practitioners identify the location of code smells, and *Smell refactoring*, in which associated sequences of refactoring strategies can be determined by a guideline summarized by Fowler et al. (1999). To preserve the consistency of functional behaviors, other software artifacts related to refactored code, such as test cases, UML diagrams, and requirement specifications (Kaur and Singh, 2019), should also be maintained. The modern Integrated Development Environments (IDEs), such as Eclipse[4] and IntelliJ IDEA,[5] embed a semi-automatic tool for applying the most commonly used refactoring strategies (e.g., *Rename*, *Extract Method*, *Replace Temp with Query*).

Prioritization. This aims to provide a proper sequence to refactor a set of code smells to achieve optimal software quality, an optional procedure for managing software quality. Code smells inevitably exist in every step of the software development lifecycle, with continuous evolution. Practitioners periodically mitigate code smells to ensure the evolvability and maintainability of software. Some code smells have sequential dependencies that require a specific order to enable further refactorings (Morales et al., 2018). With hundreds of code smells in the software, developers may not have sufficient time or resources to conduct proper refactoring to resolve most of the critical smells. In addition, the order in which a set of candidate refactoring activities should be applied can have different consequences on software quality. When facing hundreds of smells, deciding which smell should be prioritized remains a challenge for practitioners (Ouni et al., 2015). However, an improper order of refactoring may require excessive effort with unexpected consequences. To address this issue, the research community prioritizes code smells to identify a proper sequence of refactoring operations as a trade-off between software quality and practitioner concerns.

A large software system contains various types of code smells. Similarly, the same smell may occur in different places in the software (also called different contexts). Thus, smell prioritization can be divided into two categories: intra- and inter-smell prioritization (Sobrinho et al., 2021).

- Intra-smell prioritization aims to identify the most appropriate order for code entities to refactor among a set of code entities with the same type of smell.
- Inter-smell prioritization investigates which type of smell should be refactored with a high priority.

Overall, a traditional OO smell indicates a certain code pattern that can occur extensively in any OO programming language. These smells degrade the internal software quality, such as maintainability and readability, which impedes software maintenance and evolution. In contrast, Android-specific smells present performance and efficiency issues owing to framework specificities and available resources. Their occurrence is associated with the misuse of framework APIs. Although a mainstream linter (Android Lint Android, 2017) is available for detecting Android-specific issues, it is a static analyzer that cannot easily cover all paths. Moreover, the catalog of Android-specific smells should be updated with the evolution of the Android framework, and programming languages (Góis Mateus and Martinez, 2019; Habchi et al., 2019a).

3. SLR methodology

This section describes the SLR methodology. To understand the state-of-the-art and practice of Android code smells, we conducted a systematic literature review based on the guidelines provided by Kitchenham (2007). Fig. 1 presents the protocol used to conduct the SLR.

[4] https://www.eclipse.org

[5] https://www.jetbrains.com/idea

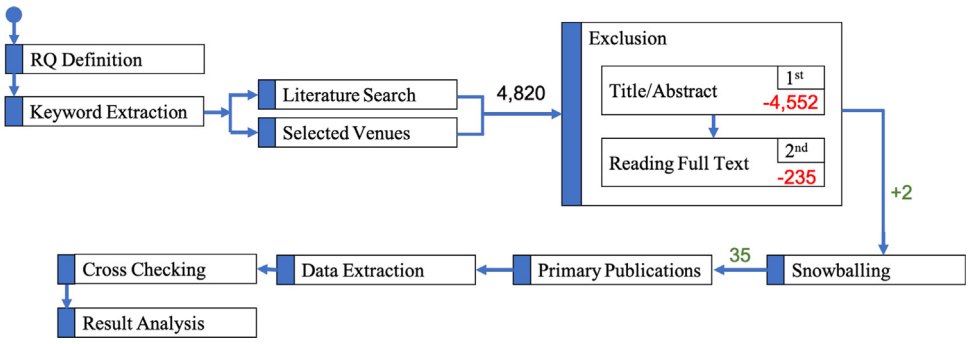


Fig. 1. Overview of SLR process.

- We defined the research questions motivating this SLR and identified the corresponding information that needs to be extracted from the literature review (cf. Section 3.1).
- We then enumerated diverse domain-related key terms that allowed us to collect the largest possible set of relevant publications from well-known publication repositories. Among a large number of collected publications, a set of exclusion criteria was defined to filter out papers that were of limited interest in our SLR. Moreover, we performed backward and forward snowballing on selected publications. The final list of papers is referred to as primary studies (cf. Sections 3.2 and 3.3).
- To ensure that the extracted findings for the given papers were complete, we pre-built a spreadsheet to contain the information (e.g., types of Android smells, code representations, and fundamental techniques) that should be extracted from papers to answer the research questions (cf. Section 3.4).

3.1. Research questions

This SLR aims to investigate the existing body of knowledge on Android code smells to understand how code smells in Android are detected and what research approaches have been proposed. Based on these goals, we define the following Research Questions (RQs):

RQ1: Which Android-specific smells are studied in the research community? This research question targets how primary studies address quality issues by detecting Android-specific code smells.

RQ1.1: What are the prevalences of studied Android smells in primary studies?

RQ1.2: Which quality attributes are affected by studied smells?

RQ2: What are the current state-of-the-art techniques for Android-specific code smells? In this research question, we analyzed the depth of research activities that researchers have conducted in the field of Android-specific code smells. To this end, we examined the following sub-questions:

RQ2.1: What are the purposes of the primary studies?

RQ2.2: What code representations are applied to analyze code smells in primary studies?

RQ2.3: Which common off-the-shelf tools are used in primary studies for generating various code representations, measuring software metrics, and profiling energy consumption?

RQ2.4: What fundamental techniques are used to analyze Android-specific code smells for different research purposes?

RQ3: What validated methods are adopted to evaluate approaches in the research community? We were interested in investigating how the research community has evaluated these approaches. For each approach, we check that the scale of the datasets used in the evaluation is available, and the aspects considered in the empirical studies are detailed.

RQ4: What are the trends in analyzing Android-specific smells in the research community? With this research question, we conducted a larger analysis of all relevant literature to investigate the evolution of research interests on Android-specific code smells over time.

Table 1
Search terms for collecting relevant literature.

Category	Terms
Program	Android; mobile; app*; smartphone*
Issue	code smell*; smell*; anti-pattern*
Research purpose	refactor*; detect*; identif*; priorit*

3.2. Search strategy

The search strategy involves searching, collecting, and distilling relevant publications from digital libraries. We now describe each step of the search strategy.

3.2.1. Search terms

Based on the research questions defined in Section 3.1, we can summarize our search terms by considering three conditions: (1) oriented target program, (2) related to software refactoring, and (3) the analysis activities of refactoring. Table 1 lists the search terms used in this study. Consequently, the search query is formed as a conjunction of three keyword conditions, as shown below:

("Android" OR "mobile" OR "app*" OR "smartphone*") AND ("code smell*" OR "smell*" OR "anti-pattern*") OR ("refactor*" OR "identif*" OR "detect*" OR "priorit*")

The asterisk character is leveraged to match all possible term variations, such as plurals. Moreover, we applied a search query to all metadata to collect more relevant publications.

3.2.2. Digital libraries

Owing to different publishers, online digital libraries have a limited number of downloadable publications. Thus, we leveraged seven well-known academic digital libraries and one publication search engine recognized as the most representative in the field of computer science to find relevant publications: including SpringerLink,[6] IEEEXplore Digital Library,[7] ACM Digital Library,[8] ScienceDirect,[9] Web of Science,[10] Scopus,[11] and Google

6 https://link.springer.com
7 https://ieeexplore.ieee.org
8 https://dl.acm.org
9 https://www.sciencedirect.com
10 https://apps.webofknowledge.com
11 https://www.scopus.com

Table 2
Inclusion and exclusion of search criteria.

Inclusion	Exclusion
Written in English	Published before 2008
Mobile-specific	Non-peer-reviewed
Research on Android apps	Irrelevant programming languages
Propose code smell tools	Republished Publications

Scholar.[12] In some cases, digital libraries do not provide a way to download large search results (i.e., metadata and full-text). Thus, we exported all search result citations from each library and implemented a Python script to download all relevant publications automatically.

Moreover, we performed a manual search to download the preprints of newly accepted papers from the latest conferences. For instance, we manually collected several accepted pre-prints from the International Conference on Software Engineering (ICSE) and International Conference on Mobile Software Engineering and Systems (MOBILESoft) in 2021 that cannot be officially indexed in the aforementioned digital libraries.

3.2.3. Publication exclusion

The search terms provided allowed us to collect an exhaustive list of publications in this domain. However, such broadness of searching also suggests that many search results may be irrelevant and beyond the scope of this SLR because of inaccurate results from search engines. Therefore, we distilled the primary studies based on the following criteria as shown Table 2. In addition, we have conducted cross-validation for each criterion to ensure that the relevant publications have been included correctly. We have shown the number of excluded papers from each stage in Fig. 1.

1. *Written in English*: English is the primary language used in academic communication. Publications that were not fully written in English were excluded.
2. *Mobile-specific*: The collected results contain many publications about code smells on traditional Java software, which is beyond the scope of this study. We thus excluded publications related to desktop software according to the titles and abstracts of the publications.
3. *Android app*: We excluded papers that analyzed desktop software related to the fifth exclusion criterion. However, the collected results contained several publications related to iOS platforms and web services due to our search term "mobile". We dismissed irrelevant publications by reading the full text.
4. *Code Smell*: In this SLR, we considered only publications that focused on bad code practices in the implementation stage of Android apps. Thus, publications that only mention code smells were excluded. For instance, FCDP (Wu et al., 2020) is excluded from this SLR because its main contribution is assessing app descriptions' fidelity rather than permission smells. We excluded test smells in our study since test smells hurt software maintainability and testing performance that end-users cannot perceive.
5. *Invalid years*: The first Android version was released in September 2008. The concept of code smell was proposed by Fowler et al. in 1999 (Fowler et al., 1999). Based on our search terms, the collected publications may contain a small portion of papers published before that time. To relieve manual effort, we directly filtered papers published before 2008.
6. *Non-peer-reviewed*: As evidence of the quality of their research, papers without peer review should be excluded from the SLR (e.g., technical reports, Ph.D. theses). However, these papers will be used to extract relevant peer-reviewed publications in the snowballing stage to avoid misses. For example, we excluded Habchi's thesis (Habchi, 2019) from the search results. However, we extracted one early access paper (Habchi et al., 2021b) from her thesis that was not obtained in our collection stage.
7. *Programming Languages*: Our search terms include "code smell" and "refactoring" to collect possible papers. Thus, the collected set contains other programming languages that do not involve the use of the Android SDK, such as CSS. Therefore, we excluded such explicit non-mobile papers.
8. *Republished Publications*: We attempted to identify republished papers for exclusion. In general, two scenarios cause such duplication: (1) the papers published in conferences were extended to a journal venue; or (2) the authors published their work with a tool or demo in the conference demonstration track. We then manually identified all questionable duplicate pairs based on author, affiliation, and content. For example, Leafactor (Cruz et al., 2017) proposed the detection of five energy-related smells in a conference. Later, the author published this paper using a detailed methodology in a journal (Cruz and Abreu, 2019). In this case, we excluded the conference paper because an extended version of the paper was already available.

3.2.4. Snowballing

As recommended in the guidelines of Wohlin (2014), we performed snowballing to confirm whether all relevant references in the collected papers were evaluated. The main aim of snowballing is to obtain additional relevant papers that may not match our search terms.

Snowballing is labor-intensive and time-consuming to perform manually. Therefore, we divided the authors into two groups. In the first group, two authors manually identified potential papers based on their references. In the second group, one author leverages ConnectedPapers,[13] a visual tool for finding similar publications based on co-citation and bibliographic coupling, to bidirectionally discover strongly relevant publications that were not yet cited in our collected set. Newly collected publications were reviewed by all authors to ensure our selection met the selection criteria.

3.3. Primary studies selection

This section elaborates on our final set of selected primary studies, as shown in Table 3. Initially, publications were collected from the seven mentioned digital libraries via the defined search query. In this phase, we only used the search engines provided by libraries to batch download the essential information of relevant publications, including the title, DOI, published venue, and authors. In addition, the searched publications were initially published in either IEEE or ACM digital libraries but were simultaneously archived on the Web of Science. Therefore, we leveraged the title and DOI to filter out duplicates within each digital library. Consequently, the dataset included 4,820 publications.

Moreover, the same papers may be collected from multiple digital libraries (Li et al., 2017), causing a double workload in future reviews. We found the following two reasons: (1) papers from Google Scholar were originally published in six other digital libraries, and (2) the same papers may exist in multiple libraries (e.g., Web of Science may contain papers published in IEEE).

[12] https://scholar.google.co.kr

[13] https://www.connectedpapers.com

Table 3
Summary of publication selection.

Steps	IEEE	ACM	Springer	Elsevier	Web of science	Scopus	Google scholar	Total
Initial search	502	102	751	99	140	2256	970	4820
Remove redundancy	502	89	704	99	128	1989	970	4481
Merge								3336
Criteria 1–2								3193
Criteria 3–5								268
Criteria 6–8								45
Final decision								33
Snowballing								2
Primary studies								35

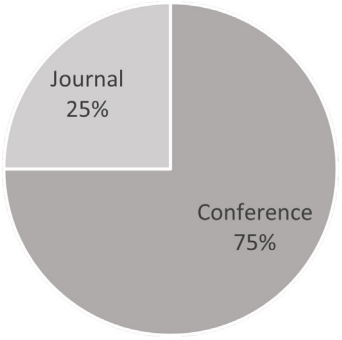


Fig. 2. Statistics of primary studies.

Fig. 3. Word cloud of all published venues of primary studies.

Thus, we merged the collected publications for six digital libraries based on DOI (a unique identifier for publications). For Google Scholar, the papers were merged with other libraries based on the title, venue, and author since we could not obtain detailed metadata from the search page.

After merging, 3336 distinct papers were retained for further review. Since not all libraries provide abstracts of papers in the downloaded metadata, we implemented a Python script to crawl them from source pages. Meanwhile, we leveraged scripts to check whether the published year and written language satisfied exclusion criteria 1–2. We then distributed the remaining papers to all co-authors. According to a review of the title and abstract, 268 out of 3,193 papers that satisfied criteria 3–5 were selected for the next review stages. We then reviewed the full content of the selected papers by applying criteria 6–8. With the final decision between the authors, we selected 33 papers as primary publications. Regarding bidirectional snowballing, 28 potential papers were collected, but only two publications satisfied our selection criteria (i.e., addressed the analysis of Android-specific code smells).

Overall, our SLR examined 35 publications as primary publications, which are listed in Table A.1. Fig. 2 shows the distribution of primary studies by publication type. 75% of our selected papers were published in conferences, which are top-ranked conferences in the domain of software engineering or co-located with these conferences. Moreover, we found that all primary studies were published in the software engineering domain, which supports our initial heuristics of considering top conferences in the software engineering domain to collect newly accepted papers.

Fig. 3 shows a word cloud of venues in which our primary studies are published. The font size of venues presents the frequency of their occurrences in this SLR. For example, MOBILESoft has the largest font size among them, which indicates that it has published most primary studies in this SLR. On the contrary, only one primary study was published in ISSTA that has the smallest font size in the figure. Most of the studies were from top conferences or journals in mobile software and software engineering (e.g., MOBILESoft, TSE, ICSE, JSS, ICSME, ASE, and EMSE), indicating that our SLR has been considered the important work from previous studies.

3.4. Data extraction

Once the primary studies were decided, we extracted the necessary information based on a taxonomy of information from primary studies to answer our research questions. Table 4 lists the relevant need to be extracted to answer the research questions. Moreover, we summarized the possible limitations of the reviewed papers that aimed to report the trends and challenges of existing works.

To mitigate the reviewer bias, all authors examined the primary studies, respectively. We then cross-validated the extracted data to answer the research questions in this SLR. If any conflict occurred in our findings, we discussed relevant issues until reaching a consensus.

4. Findings from primary studies

In this section, we summarize the findings of this SLR based on the research questions defined in Section 3.1.

Table 4
Information extracted from each primary study.

RQ	Extracted information	Description
RQ 1	Studied code smells	What type of code smells were studied in the proposed approaches?
	Affected quality attributes	Which quality attributes were affected by studied code smells?
RQ 2	Studied purposes	The approach addressed issues for Android code smells, i.e., smell detection, refactoring, and prioritization.
	Code representation	The proposed approach studied which programming language and intermediate representation, e.g., Java, Kotlin, Java bytecode, etc.
	Common tools	The approach used off-the-shelf tools to either analyze source code or measure Android code smells.
	Fundamental approaches	What fundamental approach was applied for different purposes?
RQ 3	Dataset	What dataset was used to evaluate the approach?
	Evaluation criteria	What method was applied to validate the approach?

4.1. RQ 1: Studied code smells

The first research question of our SLR relates to which Android smells were more concerned with the research community. More specifically, we aimed to understand: (1) the prevalence of Android code smells studied and (2) the quality attributes of smells affected. The following subsections present the findings of each goal.

4.1.1. Prevalence of Android code smells

In our SLR, we found that the primary studies considered 26 out of the 30 Android smells proposed by Reimann (2014), as shown in Table 5. Evidently, the *Durable Wakelock* (DW), *Inefficient Data Structure* (IDS), and *Member Ignoring Method* (MIM) smells were considered the most in our primary studies. In addition, the remaining smells were also studied in the primary studies with few times. Such results are strictly connected with the affected quality attributes considered by the authors, which are analyzed at the end of this research question. Nevertheless, Android smells such as *Bulk Data Transfer On Slow Network*, *Interrupting From Background*, and *Prohibited Data Transfer*, have not been studied in any primary study since such smells are subjective issues. The researchers lack a proper method and testing samples to evaluate their impacts. Such smells are not only introduced by developers but also depend on the end-users. For example, Bulk Data Transfer On Slow Network presents that transferring data over a slower network connection will consume much more power than over a fast connection. When the end-users run the mobile apps without a Wi-Fi connection, Android apps should use the enabled mobile network to run the functionalities. As researchers, it is hard to generate a proper criterion for detecting such smells with complex business logic. Thus, we can conclude that existing publications cannot address all Android code smells.

Furthermore, we found 17 new smells that were not introduced by Reimann (2014). To avoid inconsistent naming between studies, we applied a manual classification to compare the definitions and symptoms of these new smells with the original smells to filter out duplicates. The three aspects have been considered to identify duplicates regarding definitions and symptoms. If one smell has different symptoms (i.e., involved Android APIs) from other smells, we consider that such smell is a new smell in this domain. As a result, we identified four duplicate definitions with different names and 13 new Android smells. Table 6 presents the different names with the same definitions and symptoms in our SLR. For example, an IDS smell indicates that using HashMap in Android apps can degrade runtime performance, creating an unpredicted growth of the HashMap (Reimann, 2014). Hecht et al. renamed this smell as HashMap Usage to provide an intuitive meaning for the IDS smell (Hecht et al., 2016), which was extensively used in other studies (Góis Mateus and Martinez, 2019; Habchi et al., 2019; Marimuthu et al., 2021). In this case, we only use the term IDS in our SLR since it is an original term in this domain.

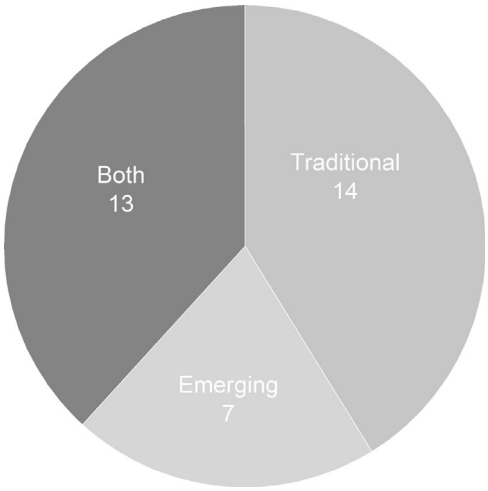


Fig. 4. Statistics of studied code smells in Android apps.

During identifying duplicate definitions, we found that 79.4% of primary studies addressed the well-known Android code smells defined by Reimann, as shown in Fig. 4. However, over half (20/35) of the examined papers introduced 11 newly introduced smells and three groups of smells in Android development, which has also been summarized in another empirical study (Prestat et al., 2022) as shown in Table 7. These emerging Android smells depict different symptoms in various aspects, such as databases, asynchronous execution, and unreasonable resource usage. In addition, 4 newly introduced smells have been addressed in AndroidLint (Android, 2017) with different detection rules to optimize the performance of Android apps (i.e., *Init onDraw*, *Recycled TypedArray*, *Unused Resources*, and *Obsolete Layout Param*). Another 6 Android smells detect the quality degradation caused by hardware usage (Hecht et al., 2015b), multi-threading (Lin et al., 2014), and resource misuse (Fatima et al., 2020; Carette et al., 2017). Moreover, we found that three groups of Android

Table 5
Traditional Android-specific code smells were identified in the previous studies.

Code Smell	Abbrv.	Freq.	Studies
Durable Wakelock	DW	15	[S01], [S10], [S11], [S12], [S14], [S17], [S18], [S20], [S21], [S25], [S26], [S27], [S28], [S32], [S33]
Member Ignoring Method	MIM	13	[S01], [S03], [S05], [S06], [S07], [S10], [S13], [S14], [S18], [S19], [S20], [S21], [S35]
Inefficient Data Structure	IDS	12	[S01], [S04], [S05], [S06], [S10], [S13], [S14], [S16], [S18], [S19], [S20], [S21]
Unclosed Closable	UC	9	[S01], [S12], [S14], [S21], [S25], [S26], [S27], [S28], [S32]
Internal Getter and Setter	IGS	9	[S01], [S03], [S10], [S13], [S14], [S16], [S19], [S21], [S35]
Leaking Inner Class	LIC	9	[S01], [S03], [S05], [S06], [S07], [S13], [S14], [S21], [S35]
No Low Memory Resolver	NLMR	7	[S01], [S03], [S05], [S06], [S14], [S21],[S24]
Leaking Thread	LT	6	[S01], [S10], [S14], [S21], [S24], [S29]
Data Transmission Without Compression	DTWC	5	[S01], [S07], [S14], [S17], [S24]
Rigid Alarm Manager	RAM	5	[S01], [S04], [S14], [S17], [S21]
Early Resource Binding	ERB	5	[S16], [S24], [S25], [S26], [S27]
Public Data	PD	4	[S01], [S14], [S21], [S24]
Overdrawn Pixel	OP	4	[S04], [S06], [S13], [S21]
Uncached Views	UV	4	[S11], [S18], [S20], [S29]
Slow Loop	SL	4	[S01], [S07], [S14], [S21]
Debuggable Release	DR	3	[S01], [S14], [S21]
Inefficient Data Format and Parser	IDFP	3	[S01], [S14], [S21]
Inefficient SQL Query	ISQLQ	2	[S01], [S14]
Set Config Changes	SCC	2	[S14], [S21]
Nested Layout	NL	2	[S15], [S21]
Unnecessary Permission	UP	2	[S02], [S23]
Not Descriptive UI	NDUI	1	[S21]
Network & IO Operations in Main Thread	NIOMT	1	[S15]
Dropped Data	DD	1	[S21]
Untouchable	UT	1	[S21]
Uncontrolled Focus Order	UFO	1	[S21]

Table 6
Duplicate namings.

Original name	New name
Inefficient Data Structure	HashMap Usage
Overdrawn Pixel	UI Overdraw
Uncached Views	View Holder
Init onDraw	Draw Allocation

code smells were proposed for specific quality attributes or components, and each group contained a set of smells. Thus, we identified them as groups and elaborated them as follows.

Security smells: Ghafari et al. summarized 28 *security smells* in Android apps by conducting an extensive literature review (Ghafari et al., 2017). Security smells are recurring code patterns that signal the prospect of security weaknesses. Even if such smells do not always cause security breaches, they deserve attention and inspection (Rahman et al., 2019). Security smells involve various components in Android apps, such as abuse of Android components, secure development practices, and insecure data operations. Among the 28 security smells, we identified 25 distinct smells. In other words, three smells were defined in previous studies (i.e., *Unnecessary Permission* (UP), *Debuggable Release* (DR), and *Tracking Hardware ID*). Moreover, the concept of security smells was leveraged in other domains, such as Infrastructure as Code (IaC) (Rahman et al., 2021; Rahman and Williams, 2021) and Python gists (Grano et al., 2017).

SQL smells: Due to the limited hardware resources in Android devices, improper SQL statements to operate records in the local database may cause extensive overhead, which severely affects the responsiveness of Android apps. Lyu et al. (2019) conducted a literature review and summarized 52 anti-patterns[14] related to SQL statements that may lead to battery drain and performance decline in Android apps. SQL smells are not specific to Android. They also exist in many popular mobile OS (e.g., iOS and Windows Phone).

Presentation Layer Smells: The aforementioned Android smells exist in the functional source code or AndroidManifest.xml files. Therefore, Carvalho et al. (2019) proposed 20 new Android smells to identify bad practices on Android UI components (such as Activities and Fragments) and different types of resources (such as layouts, strings, and images). Such smells impact the maintainability and evolvability of apps and degrade performance and user experience.

Summary for RQ1.1. *Durable Wakelock*, *Inefficient Data Structure*, and *Member Ignoring Method* are the most-considered smells in the primary studies. 86.7% of code smells from catalogs

[14] https://github.com/USC-SQL/SQLABenchmark

Table 7
Newly introduced android-specific smells in literatures.

Code Smell	Abbrv.	Type[a]	Freq.	Studies
Init onDraw	IOD	I	5	[S04], [S06], [S11], [S18], [S20]
Heavy Broadcast Receiver	HBR	I	4	[S05], [S13], [S14], [S24]
Heavy AsyncTask	HAT	I	4	[S05], [S09], [S24], [S31]
Heavy Service Start	HSS	I	3	[S04], [S14], [S24]
Recycled TypedArray	RTA	I	3	[S11], [S18], [S20]
Unused Resources	UR	I	3	[S25], [S26], [S27]
Unsupported Hardware Acceleration	UHA	I	2	[S06], [S24]
Obsolete Layout Param	OLP	I	2	[S11], [S18]
Excessive Method Calls	EMC	I	2	[S18], [S20]
Security Smells	SS	G	2	[S08], [S22]
Presentation Layer Smells	PLS	G	2	[S15], [S34]
Useless Parent Layout	UPL	I	1	[S29]
Unsuited LRU Cache Size	ULRU	I	1	[S06]
SQL Smells	SQLS	G	1	[S30]

[a]G denotes a group of Android smells that consists of multiple smells. I denotes independent smell.

Table 8
The affected qualities of Android-specific code smell.

Affected qualities	Smells
Energy efficiency	DTWC, DW, ERB, RAM, HAT, UC, DBS
Memory efficiency	IGS, LIC, LT, NLMR, SCC, HSS, HAT, UCS, OLP, EMC, DBS
Security	DR, PD, UP, SS
Performance	NL, MIM, IGS, IDFP, ISQLQ, IDS, RAM, SL, OP, UV, HBR, IOD, HSS, UHA, RTA, UR, UPL, PLS
Accessibility	UT, UFO, NDUI

(Reimann, 2014) have been studied. The research community has suggested new synonyms for the existing Android smells. This indicates the fragmentation of definitions owing to the lack of systematic taxonomies. In addition, we found that the community continuously proposed new code smells from various perspectives.

4.1.2. Affected quality attributes

Android code smells affect the different quality attributes of an app. We extracted the affected quality of each smell to understand the quality attributes of Android apps in the research community. Thus, we integrated the initial definitions from Reimann (2014) and evaluated results of the primary studies to identify the affected qualities of Android smells. To avoid the bias among authors, we have re-assigned the affected qualities to each smell based on the initial definitions and objectives in the paper. Consequently, we categorized all Android code smells into five quality attributes, as shown in Table 8. We describe each quality attribute as follows.

Energy Efficiency. Energy consumption has been an issue for a long time when using mobile devices. Although advanced hardware and battery technology have maximized the capacity of Android devices for energy consumption, these improvements cannot prevent the unnecessary battery drain of Android devices when apps incur inefficient usage on CPU/GPU and sensors (Li and Halfond, 2014). For example, *Durable Wakelock* is the most studied smell, as shown in Table 5. As shown by Khan et al. (2021b), an app may acquire a wake lock to execute tasks in the foreground. If the wake lock cannot be released at the proper time, it can drain the battery of the device and affect the lifespan of the battery. Similarly, if the sensors cannot be closed in time, this leads to energy inefficiency (i.e., *Unclosed Closable*) (Banerjee et al., 2017).

Memory Efficiency: Unlike desktop software, a trivial process in Android apps may lead to memory shortage due to its limited capacity to run program (Soh et al., 2016). For instance, the HashMap[15] is a typical data structure in Java. Using HashMap entails an auto-boxing process, where primitive types are converted into generic objects (16 bytes) that are larger than primitive types (4 bytes) (Habchi et al., 2019a). Thus, the Android framework recommends replacing HashMap with SparseArray, which is more memory-efficient. Optimizing the data structure and code design in algorithms is required for Android apps to improve memory efficiency while operating on large-scale data.

Security: Similar to traditional software, Android apps also face security risks due to bad designs. In particular, permission and Inter-Component Communication (ICC) mechanisms are the pillars that protect data security and privacy in Android apps. According to an empirical study (Wu et al., 2021), 54.8% of Android apps among 60,846 apps over-claimed an excessive number of permissions to accelerate development without being removed in the released version (i.e., *Unnecessary Permissions*), which leads to security risks. This category mainly contains API misuses that access sensitive data on the devices.

Performance: Performance directly reflects whether Android apps execute smoothly during runtime, such as launch time and loading duration. Cruz and Abreu (2017) indicated that resolving Android code smells related to performance also improves energy and memory efficiency.

Accessibility: Android apps serve all types of potential end-users. Accessibility ensures that end-users with limited vision or other physical impairments can use Android apps. Thus, visual elements in mobile apps, such as buttons and labels, should be 48dp or larger (Android, 2012). Specifically, blind users interact with mobile devices using a built-in screen reader (i.e., TalkBack[16]). However, the prerequisite for using such services is that natural language labels should be added to every UI element. Unfortunately, more than 77% of Android apps lack such labels (i.e., *Not Descriptive UI*) (Chen et al., 2020).

As shown in Table 8, performance concerns are the focus of well-known code smells in Android apps. Fig. 5 shows the statistics of affected qualities from our primary studies according to the mapping of code smells to qualities. Performance, energy, and memory efficiencies are the most popular concerns in this domain. In addition, unlike OOP code smells, these non-functional quality attributes produce various symptoms in Android apps, such as battery consumption (Khan et al., 2021a), long loading duration (Banerjee et al., 2014), and reduced responsiveness or smoothness of the user interface (Carvalho et al., 2019). These symptoms directly impair the user experience since end-users

15 https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
16 https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback

Table 9
Analysis purposes of Android-specific code smell in the literature.

Study	Detection	Refactoring	Prioritization	Study	Detection	Refactoring	Prioritization
S01	✓			S19	✓	✓	
S02	✓			S20	✓		
S03	✓			S21	✓		
S04	✓	✓		S22	✓		
S05	✓			S23	✓		
S06	✓			S24	✓	✓	
S07	✓			S25	✓		
S08	✓			S26	✓		
S09	✓	✓		S27	✓		
S10		✓		S28	✓		
S11		✓		S29	✓		
S12	✓	✓		S30	✓	✓	
S13	✓			S31	✓	✓	
S14	✓	✓		S32	✓	✓	
S15	✓			S33	✓		
S16	✓	✓	✓	S34	✓		
S17	✓			S35	✓		
S18	✓	✓					
Total					33	13	1

can perceive them with limited hardware resources in Android devices.

Summary for RQ1.2. Performance, energy, and memory efficiency-related smells have been studied frequently to improve the user experience. This is because bad code practices may cause resources to not be effectively utilized to achieve maximum performance. Compared to traditional software, Android apps were proven to have performance sensitivity due to limited hardware resources.

4.2. RQ 2: State-of-the-art techniques

We now investigate how the analyses described in the literature have been implemented. In particular, we analyzed the purposes of the analyses (Section 4.2.1), code representations leveraged to address code smells (Section 4.2.2), supported tools applied (Section 4.2.3), and fundamental approaches in the studies (Section 4.2.4).

4.2.1. Activities of analyses

Code smells have been applied to address various issues. To identify the recurring purposes for Android-specific smells, the objectives of each literature have been classified based on their leveraged methods and analyzed results. Finally, we identified three recurring study purposes in this domain. Table 9 classifies the primary studies based on their respective purposes. Smell detection appeared to be the most considered, with 33 primary studies taking it into account. This finding is understandable since detection is the initial step in remedying code smells. Moreover, 37.1% (13 out of 35) considered refactoring, while only one of the primary studies involved code smell prioritization.

Similar to the OOP smell detection, the most existing detection techniques applied static analysis to detect Android smells. Currently, dynamic analysis is majorly applied to estimate energy consumption in this domain. In addition, the existing detection techniques are still immature, particularly in terms of smell coverage and compatibility with new versions of Android. Nevertheless, locating bad practices in source code is essential for the automatic refactoring of code smells. The research community still makes many efforts in smell detection, which naturally makes refactoring and prioritization less attended.

Summary for RQ2.1. Most primary studies support the detection of code smells in Android apps with static analysis techniques. Prioritizing code smells has been the least considered by the Android research community.

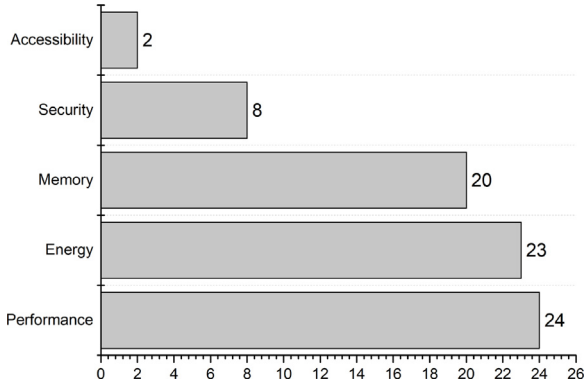


Fig. 5. Statistics of affected qualities addressed in the literature.

4.2.2. Code representations

In this section, the underlying code representations used in this study are discussed. Table 10 enumerates the details of adopting the different code representations in the primary studies. The source code used for analyzing Android smells is the most applied target (65.7% of primary studies), while 50% of primary studies support Android APK files as input for smell detection. Among them, 5 primary studies supported the analysis of both source code and APK files.

As shown in Table 10, Java is the most popular code representation for analyzing Android smells. The existing approaches that analyze APK files often decompiled Android apps and translated app bytecode to other Intermediate Representations (IRs). These IRs can be applied to generate an AST and extract software metrics using existing tools (Li et al., 2017). Among these, Jimple is the most commonly used IR in our primary studies. As we know, the smell presents a potential symptom in the source code. However, existing studies have also considered Android APK files to identify refactoring opportunities. According to our primary studies, we summarize two major reasons for detecting code smells from APK files as follows.

Compatibility of programming languages. Java is a key pillar in Android development. Since Google I/O 2017, Kotlin[17] has been supported as an alternative language for Android apps. Kotlin is a statically typed programming language that runs on the Java

[17] https://kotlinlang.org

Table 10
A summary of code representations used in the primary studies.

No.	Target		Code representation					
	Source	APK	Java_Class	Java	Kotlin	Jimple	Smali	WALA
S01	✓			✓				
S02		✓		✓				
S03		✓				✓		
S04	✓	✓		✓				
S05	✓	✓				✓		
S06	✓			✓				
S07	✓			✓				
S08	✓			✓				
S09	✓			✓				
S10	✓			✓				
S11	✓			✓				
S12	✓					✓		
S13	✓					✓		
S14	✓	✓		✓		✓		
S15	✓			✓				
S16	✓			✓				
S17	✓			✓	✓			
S18	✓			✓				
S19	✓	✓				✓		
S20	✓			✓				
S21	✓			✓				
S22	✓	✓		✓				
S23		✓		✓				
S24	✓			✓				
S25		✓	✓					
S26		✓	✓					
S27		✓	✓					
S28		✓	✓					
S29		✓				✓		
S30		✓				✓		
S31		✓						✓
S32	✓			✓				
S33		✓					✓	
S34		✓		✓				
S35	✓			✓				
Total	23	17	4	22	1	8	1	1

Virtual Machine (JVM). Kotlin's syntax aims to increase productivity by reducing the amount of boilerplate code. Rough estimates indicate an approximate 40% cut in Lines of Code (LOC) (Flauzino et al., 2018). In addition, it is entirely interoperable with Java. It is possible to mix Java and Kotlin in the same Android app. With these benefits, the adoption of Kotlin for Android development has experienced explosive growth since 2017. In January 2022, 85.67% of the top 500 apps in the Play Store were written in Kotlin Anon (2022a). However, there is a lack of off-the-shelf tools to analyze Kotlin source code. To overcome this, Mao et al. (2020) and Goaër (2020) (i.e., S14 and S17) decompiled Android APK files to obtain Java bytecode or another IR.

Energy Estimation. As the finding in RQ1.2 indicates, smells related to performance and energy efficiency have received more attention from research communities. Generally, such smells indicate a specific issue, i.e., battery drains, but lack the context to locate these symptoms in the source code (Anwar, 2020). The smells caused by API misuses can be detected by API signatures using static analysis, such as *Durable Wakelock* and *Unclosed Closable*. However, other smells that affect performance and energy consumption cannot be identified with a certain code pattern in the source code since the desktop software has sufficient resources to handle the trivial overhead from such code smells. Running Android apps on emulators or devices is necessary to measure the energy and memory consumption of specific functionalities. This is the primary reason why the research community has also considered Android APK files to identify code smells.

Summary for RQ2.2. Java is the most adopted code representation for analyzing code smells in Android apps. Moreover, some studies leverage Android APK files to overcome the lack of support tools for Kotlin and precisely measure energy consumption at runtime.

4.2.3. Off-the-shelf tools to support analysis

To answer this question, we tracked the tools used in primary studies. Based on different purposes, we identified four types of recurrently used support tools, as shown in Table 11. These tools are often off-the-shelf components that implement common analysis processes.

We observed that the proposed approaches that analyze code smells from Android APK files often leverage various off-the-shelf tools to obtain another IR. IR is a simplified code format that represents the original Dalvik bytecode and facilitates processing since Android Dalvik is known to be complex and challenging to manipulate. The proposed approaches also applied different off-the-shelf tools to construct diverse program flow graphs (i.e., AST and call graphs) based on the obtained IR. The Soot framework is considered the most adopted tool to generate a program graph. Among the listed tools, UAST is unique in that it constructs AST from Kotlin source code, but it has only been applied in two studies: S08 and S17. Android apps also contain XML files to describe essential information about apps (i.e., `AndroidManifest.xml`) and activity layouts, unlike traditional OO programs. For this characteristic, the existing studies leverage various XML parsers to analyze component declarations and bad practices in UI layouts, such as over-claimed permissions (Scoccia et al., 2019) and deprecated resource usage (Gadient et al., 2019).

One remark is that an energy profiler based on software/hardware is applied to measure energy consumption. In general, such measurements require running apps on a physical device or

Table 11
List of recurrent supported tools for analyzing Android-specific smells.

Type	Tool	Description	Studies
Intermediate representation	AndroGuard (Desnos, 2012)	A tool to decompile and analyze Android apps	S33
	Apktool	A tool for decompiling Android apps	S02, S23
	dex2jar	A DEX to Java bytecode translator	S02, S23, S25, S26, S27
	Dexpler (Bartel et al., 2012)	A DEX to Jimple translator	S03, S05, S19, S30
	JADX[a]	A DEX to Java source code translator	S22
	JD-Core-java	A tool to translate Java bytecode to source code	S02, S23
Syntactical parser	Soot (Lam et al., 2011)	A Java optimization framework	S03, S05, S12, S14, S19, S29, S30
	WALA[b]	A Java static analysis framework	S31
	JavaParser[c]	A parser to generate AST from source code	S15, S21, S22
	JDOM[d]	A tool to parse XML file	S15
	JDT[e]	A parser to generate AST from source code	S01, S09
	XmlScanner	A library to parse XML file	S08, S17
	Spoon (Pawlak et al., 2016)	A library for generating AST from Java source code	S06, S19
	UAST[f]	A library for generating AST for different JVM languages	S08, S17
Energy Profiler	Monsoon (Marimuthu et al., 2021)	A physical device to measure power consumption of Android apps	S12, S30
	Naga viper	A physical measurement device and monitors energy-related metrics	S19
	Trepn profiler[g]	An energy profiler for Qualcomm-based Android devices	S18
Software metrics	Scitool understand[h]	A tool to compute software metrics	S07
	SonarQube (Campbell and Papapetrou, 2013)	A tool to compute software metrics	S20
	Deignite (Sharma et al., 2016)	A tool to compute software metrics	S11

[a] https://github.com/skylot/jadx
[b] http://wala.sourceforge.net
[c] https://javaparser.org
[d] http://www.jdom.org
[e] https://www.eclipse.org/jdt
[f] https://plugins.jetbrains.com/docs/intellij/uast.html
[g] https://developer.qualcomm.com/forums/software/trepn-power-profiler
[h] https://www.scitools.com

emulator. Moreover, these studies also generated a sequence of actions that provided inputs to the apps based on UI elements and automatically executed each sequence multiple times to obtain accurate results with the help of MonkeyRunner.[18] In addition, we found that some empirical studies also leverage other profilers excluded from the table to investigate the impacts of energy smells in Android apps. For instance, Palomba et al. (2019) analyzed the impacts of energy-related smells and discovered that their presence notably impacts battery usage using PETrA (Di Nucci et al., 2017a), a software-based tool for estimating energy consumption in Android apps. Compared to hardware-based profilers, software-based profilers are compatible with all smartphones, but require tuning the hyperparameters for different devices (Di Nucci et al., 2017b).

Regarding software metrics, various off-the-shelf tools are applied to extract some traditional OO metrics (e.g., LOC) to facilitate the analysis of Android-specific smells.

Summary for RQ2.3. The research community has applied various off-the-shelf tools to convert Android APK files and source code into IRs. Among them, the Soot framework is the most adopted tool for static analysis. Unlike OO programs, existing studies also leverage various profilers to measure energy consumption to identify relevant smells.

4.2.4. Fundamental studied approaches

Since Android-specific smells indicate performance-oriented issues, the addressed methods and features are different from those of OO smells. In our study, we identified six fundamental methods that are often adopted in conjunction in the literature.

Table 12 summarizes the different methods used in the reviewed publications. The summary statistics show that the rule-based method used for detecting and refactoring Android smells is the most applied method (19 of the primary studies, i.e., 54.2%). Another four methods (metric-, learning-, history-, and search-based) have been considered for addressing smells in a small portion of our primary studies. In particular, we found that 32.3% of the primary studies involved program analysis to detect Android smells according to energy consumption and API usage. We identified three types of program analyses.

[18] https://developer.android.com/studio/test/monkeyrunner

Table 12
Summary of different fundamental methods applied in primary studies.

Methods	Studies	Count
Rule-based	S01, S02, S08, S10, S11, S14, S15, S17, S18, S19, S20, S21, S22, S24, S29, S30, S31, S32, S34	19
Metric-based	S03, S06, S19	3
Learning-based	S05, S07, S33, S34, S35	5
History-based	S06	1
Search-based	S13, S16, S04	3
Model/Program analysis	S09, S12, S22, S23, S25, S26, S27, S28, S30, S31, S32	11

Taint analysis. A taint analysis is a type of information flow analysis in which objects are tainted and tracked using a data flow analysis. For example, Statedroid (Xu et al., 2018) combined type checking and taint analysis to detect open-but-not-used resources (i.e., *Unclosed Closable*). It generates a CFG to check whether the states of the resource instances violate pre-defined rules. As another example, M-Perm (Chester et al., 2017) leverages static taint analysis to determine whether the relevant API usages of dangerous permissions are invoked in the developer's code (i.e., *Unnecessary Permissions*).

Model checking. Model checking is the process of verifying whether a finite-state system satisfies a given specification. For example, GreenDroid (Liu et al., 2014b) utilized dynamic information flow analysis to detect the missing deactivation of sensors and wake-locks in Android apps. GreenDroid executes Android apps on Java PathFinder (JPF) (Visser et al., 2004), a system for verifying executable Java bytecode programs and exploring the states of the sensors. Then, it analyzes the registration of sensors and wake locks for each explored state to determine energy-related smells using dynamic taint analysis.

Symbolic execution. Symbolic execution is useful for generating possible program inputs and detecting infeasible paths. Symbolic values are typically considered as inputs to propagate execution. These symbolic values are used to generate expressions and constraints that can be further leveraged to produce possible inputs fulfilling all conditional branches inside the given path. These inputs can then be used as test cases to explore the given path for a repeatable dynamic analysis. If no input is produced, the given path is confirmed to be infeasible. For example, Banerjee and Roychoudhury (2016) leveraged symbolic execution to generate design expressions for sensor-related resource usage, such as acquiring and releasing wake locks. The intersection between the design and predefined defect expressions negates energy-efficiency guidelines.

Moreover, our primary studies applied two fundamental methods to their solutions. Fig. 6 shows the co-occurrent relationship among the fundamental methods. In our reviewed papers, the search-based method was able to address both detection and refactoring in this domain. Other methods generally focus on one activity (detection or refactoring). As we can see, the rule-based method is the most popular and is integrated into other methods to provide additional supplementary information for detection and refactoring. For instance, EnergyPatch (Banerjee et al., 2017) identified program paths containing energy smells using a model checking method. It then utilizes symbolically pre-defined rules to automatically generate repair expressions.

Although Android apps are also written in OO languages, they have different characteristics in their functionalities. Table 13 summarizes all features used in primary studies for addressing Android-specific code smells, which consist of general features in OO and some characteristics in the Android framework. We only describe the differences in Android compared to the detection of traditional OO smell as follows.

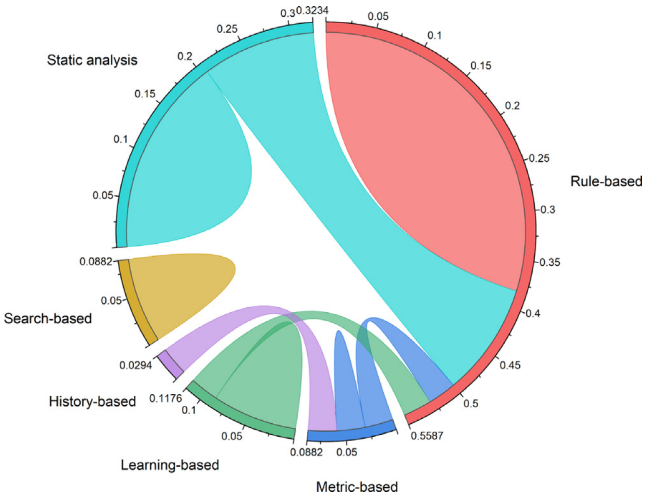


Fig. 6. Co-occurrent relationship among adopted methods.

Abstract syntax tree. An AST is a tree-based data structure that represents the structure of a program code that contains semantic and structural information. It is widely applied in detecting OO smells such as code clones (Lei et al., 2022). In this study, 17 primary studies (48.5%) take into account ASTs for smell detection, which is the most popular general feature.

Software metric. In addition to ASTs, several software metrics are used in this domain to measure the properties of Android apps. However, the metrics used in this domain are different from those of the traditional OO program because of Android characteristics (e.g., activity, services, and broadcast receiver). For example, Fakie (Rubin et al., 2019) and Paprika (Hecht, 2015) combined IsActivity (denotes if a class is an activity) and ownOnLowMemory (denotes classes with the `onLowMemory` method) metrics to detect NLMR. Such metrics have not been used to detect traditional OO smells. In particular, researchers have proposed new metrics based on their demands. In another example, Hot-Pepper (Carette et al., 2017) leverages Naga Viper to dynamically measure energy-related metrics, such as the joule consumption of Android apps, to detect three smells (i.e., IDS, IGS, and MIM). These metrics assist the proposed approach in automatically seeking optimal refactoring opportunities. In this SLR, 12 primary studies explicitly leveraged metrics to address Android smells.

Flow graph. A program flow graph is a graph-based representation that provides various information flows to researchers for analyzing software programs, such as control flow, call, and program dependency graphs. The use of a flow graph for detecting Android smells is the same as that for detecting traditional smells. 9 studies take into account various flow graphs as features.

Commit. Generally, the research community leverages the characteristics of historical commits (i.e., change-proneness) to localize smell instances. However, change-proneness cannot provide useful information for detecting Android-specific smells since they present bad practices of misusing APIs (Android framework or Java). For example, Sniffer (i.e., S06) (Habchi et al., 2019a) integrated commits from the change history of open-source apps to investigate the evolution of Android smells. We found only 2 studies that addressed this issue. The main challenge of commit analysis is the lack of open-source Android apps.

API identifiers. As mentioned, most Android-specific smells originate from API misuse. In other words, given an Android

Table 13
List of primary studies according to approach and Android characteristics.

Study	General OO				Android Characteristics			
	AST	Metrics	Flow-graph	Commit	API	Manifest	UI layout	Event Callbacks
S01	✓				✓			
S02					✓	✓		
S03		✓						
S04	✓	✓	✓		✓	✓		
S05		✓						
S06	✓	✓		✓				
S07		✓						
S08	✓				✓	✓		
S09	✓		✓		✓			
S10	✓				✓			
S11	✓				✓		✓	
S12	✓				✓			✓
S13		✓			✓	✓		
S14	✓	✓						
S15		✓			✓	✓	✓	
S16		✓			✓			✓
S17	✓				✓	✓		✓
S18	✓				✓			
S19	✓	✓			✓			
S20	✓	✓			✓			
S21	✓	✓			✓	✓	✓	
S22	✓				✓			
S23			✓		✓	✓		
S24	✓				✓			✓
S25					✓			✓
S26					✓		✓	✓
S27					✓		✓	✓
S28			✓		✓			✓
S29			✓		✓		✓	✓
S30			✓		✓			✓
S31			✓		✓			✓
S32			✓		✓		✓	✓
S33			✓	✓	✓	✓		
S34							✓	
S35	✓				✓			
Total	17	12	9	2	29	9	8	12

smell, it probably indicates specific API usage. Thus, an API identifier is the best way to narrow the scope of smell detection. For example, aDoctor (Palomba et al., 2017) leverages regular expressions to match exact identifiers for detecting smells. We found 29 primary studies (82.8%) in this SLR that adopted API identifiers to detect Android smells.

AndroidManifest. `AndroidManifest` is an essential XML file in Android apps that consists of various necessary information such as permissions, activities, and services. For example, Khan et al. (2021b) directly searched for the `WAKE_LOCK` permission from the `AndroidManifest` file to directly exclude apps that do not use wake lock. In this SLR, 9 primary studies extracted information from the `AndroidManifest` file.

UI layout. The structure of user interfaces of Android apps is defined by layouts, which can be declared in either XML files or dynamically generated in Java code. Eight studies in this SLR considered XML layouts to support a complete analysis. Most of these studies directly analyzed the layout to detect smells that impact accessibility, including UFO and Untouchable. However, Android also allows developers to generate layouts from source code. In this case, parsing XML files can no longer detect relevant smells. In particular, UIS-Hunter (Yang et al., 2021) took screenshots of UI layouts to detect smells that degrade accessibility rather than parse XML files.

Event callbacks. A number of callbacks are used in Android to handle various events, including lifecycle and UI event callbacks. These callbacks have neither connections among them nor direct connections with the app code. Twelve studies considered event callbacks for smell detection. They utilized these callbacks to explore the states of sensors and wake locks via static program analysis.

Summary for RQ2.4. The most commonly applied method is based on rules for addressing Android code smells. Most studies have considered multiple features for detection. APIs and ASTs have been primarily taken into account by the research community.

4.3. RQ 3: Dataset

We now investigate whether the existing approaches are publicly available to the Android community and how they validate the performance of Android-specific smells.

4.3.1. Public tools

Table 14 summarizes publicly available tools, among which 23 (67%) studies provided open-source tools for the Android community. Among these tools, GreenDroid (Liu et al., 2014b) (i.e., S26) requires end-user requests via email. In addition, we found that only a small proportion of studies integrated their tools into Android Lint. As shown in Fig. 7, the mean number of smells studied was 6 in our primary studies. Only DAAP (Rasool and Ali, 2020) (i.e., S21) implemented a method to analyze a maximum of 25 Android bad smells. Compared to the Android smells listed in RQ1.1 (cf. Section 4.1.1), such a smell coverage of existing approaches cannot provide sufficient capacity to analyze code smells in Android apps. In this case, practitioners have to leverage several tools to detect bad Android smells as much as possible. However, this is a non-trivial task for developers.

Summary for RQ3.1. Most studies have provided tools for the research community to conduct further research. In addition, most studies were able to detect up to six types of Android smells.

Table 14
List of primary studies with publicly available tools, and information on evaluation.

No.	Tool	Publicly available?	Evaluation		
			# smells	# App	Source
S01	aDoctor	✓	15	18	–
S02	P-Lint	✓	1	40	–
S03	Paprika	✓	8	15	–
S04	Randroid	✓	5	52	–
S05	Fakie	✗	10	3041	AndroZoo
S06	Sniffer	✓	8	324	F-Droid
S07	Gupta et al.	✗	4	10	Github
S08	Gadient et al.	✓	1[a]	732	F-Droid
S09	AsyncDroid	✓	1	9	Github
S10	Iannone et al.	✓	5	–	–
S11	Leafactor	✓	5	140	F-Droid
S12	Banerjee et al.	✗	2	10	F-Droid
S13	MOGP	✗	10	184	GitHub
S14	Droidlens	✓	18	20	Github
S15	AndroidUIDetector	✓	15	619	F-Droid
S16	EARMO	✓	8	20	F-Droid
S17	GreenCheck	✓	3	–	–
S18	Chimera	✓	8	609	MUSE
S19	Hot-Pepper	✗	3	5	F-Droid
S20	E-Debitum	✓	7	3	Github
S21	DAAP	✓	25	7	–
S22	Jandrolyzer	✓	8	2890	F-Droid
S23	M-Perm	✓	1	500	Play Store
S24	xAL	✓	12	9	F-Droid
S25	E-GreenDroid	✗	4	9	F-Droid
S26	GreenDroid	✓	4	13	F-Droid
S27	NavyDroid	✗	4	17	F-Droid
S28	statedroid	✗	2	220	F-Droid
S29	PerfChecker	✓	2	29	–
S30	RAT-TRAP	✓	1	1887	Play
S31	Asynchronizer	✗	1	13	–
S32	EnergyPatch	✓	2	12	F-Droid
S33	Khan et al.	✗	1	200	Github
S34	UIS-Hunter	✗	1[a]	9286	Rico
S35	ACS-TNN	✓	3	578	MUSE

[a]These researches can detect one group of Android smells.

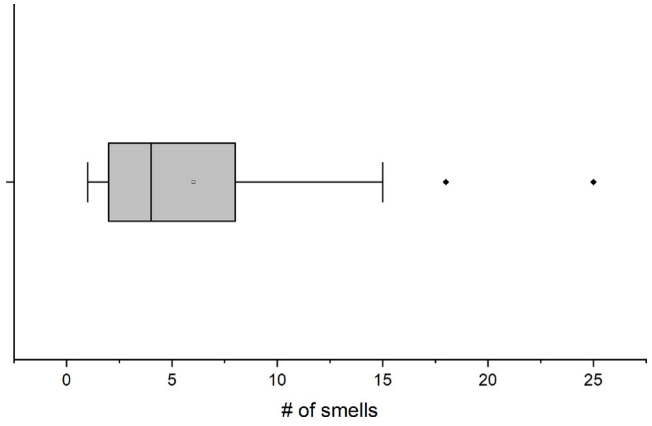


Fig. 7. Distribution of the number of studied smells in primary studies.

4.3.2. Evaluation criteria

To investigate this question, we extracted the number of evaluated apps, benchmarks, and app sources from our primary studies. As shown in Table 14, most studies conducted their evaluation on small-scale open-sourced repositories, including AndroZoo (Allix et al., 2016), GitHub, and F-Droid.[19]

We also observed that most primary studies manually identified a small set of code smells as a golden set from open-sourced apps since a benchmark for bad Android smells is lacking. For example, Palomba et al. (2017) evaluated the detection results of their approach using a manually built oracle of code smells that required approximately 180 man-hours. Khan et al. (2021b) also manually inspected the code snippets of commit histories to identify the misuse of wake lock from 32 apps. Habchi et al. conducted an empirical study to validate the acceptance of detected smells from practitioners by analyzing commits.

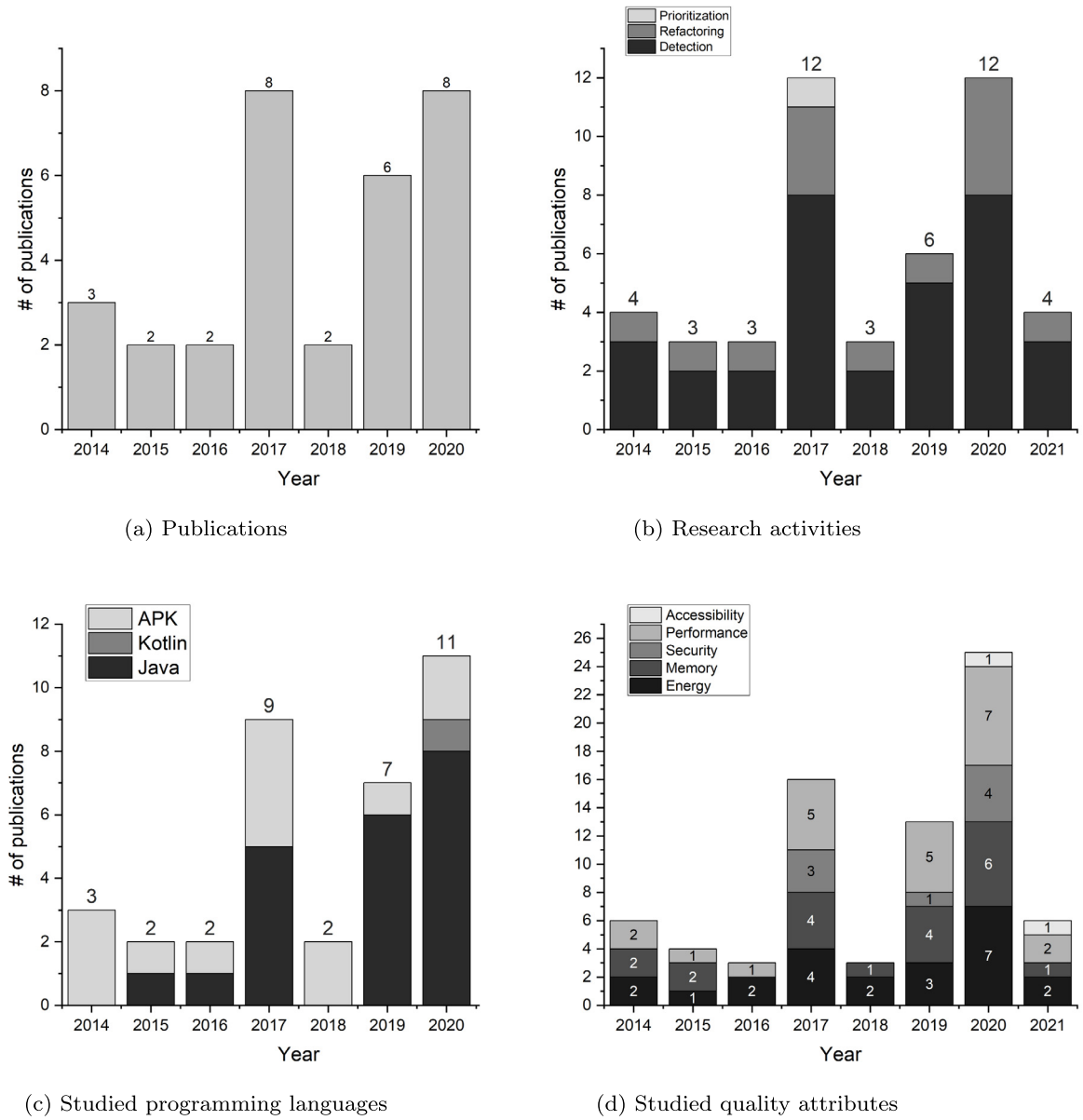
Other studies (Habchi et al., 2019; Gupta et al., 2019) utilized existing tools (e.g., aDoctor Palomba et al., 2017 and Paprika Hecht et al., 2015a) to collect instances of code smells from their own Android repositories as benchmarks. Although these well-known tools can achieve excellent performance in detecting smells, the evaluated approaches inherit all the known limitations of previous studies, such as smell coverage and detection performance. In particular, the proposed smells can be deprecated by the evolution of Android, such as Internal Getter/Setters (Habchi et al., 2019a). Alternatively, the API signatures can be changed for different Android API levels (Khan et al., 2021b).

Summary for RQ3.2. Some of the primary studies adopted the detection results from previous studies as benchmarks, including the known limitations of previous studies (i.e., detection performance and smell coverage).

4.4. RQ 4: Trends analysis

Although Android was released in 2008, research on code smell has flourished in recent years. We investigated general trends in the research according to various aspects.

[19] https://f-droid.org



(a) Publications (b) Research activities

(c) Studied programming languages (d) Studied quality attributes

Fig. 8. Trends related to Android-specific code smells.

Fig. 8 presents the distribution of publications from our dataset based on their published years. Android code smells were first proposed by Reimann et al. in 2014 (Reimann and Aßmann, 2013). Although Android code smells are relatively young compared to other domains, such as refactoring in OO programs or Android testing, significant contributions that satisfy our criteria have been published since 2014. As time goes by, researchers have become increasingly active in this domain. As shown in Fig. 8(a), the annual publications are still fewer than in other directions in Android. In addition, we observed that many papers excluded in our primary studies leveraged existing detection tools to conduct empirical studies on various aspects, such as the comparative studies between OO and Android smells (Oliveira et al., 2018; Hamdi et al., 2021), diffusion and impact of Android smells (Habchi et al., 2021b; Carette et al., 2017; Das et al., 2016; Ghari et al., 2019), and smell impacts in different programming languages (i.e., Java and Kotlin) (Flauzino et al., 2018; Ardito et al., 2020). For instance, Das et al. (2020) applied AndroidLint to investigate the statistical differences in performance-related smells and awareness of Android developers. This indicates that practitioners were aware of the performance issues detected by AndroidLint. In addition, similar to traditional OO smells, not all Android smells can be remedied before release owing to various factors, such as delivery time and budget. A prioritization method is required to assist developers in eliminating critical smells.

Over time, the research community has dedicated more effort toward detecting and refactoring Android smells (cf. Fig. 8(b)). However, the prioritization of Android smells has not been considered in our primary studies. We further examined the programming language challenge for Android smells (cf. Fig. 8(c)). Java is the mainstream target for analyzing code smells in Android apps since various existing tools support Java analysis. Although Kotlin has been extensively applied in Android development in recent years (Oliveira et al., 2020), existing studies prefer to convert Kotlin into Java by building APK files, owing to the lack of off-the-shelf tools. In addition, Fig. 8(d) shows a steady increase in publications addressing performance-, memory-, and energy-related smells. Since 2017, Android-specific smells have received more attention from the research community.

Summary for RQ4. Research addressing Android code smells remains immature. The yielded approaches need to consider

more programming languages and research studies (i.e., refactoring and prioritization).

5. Discussion

The findings yielded by investigating the research questions of this SLR constitute many discussion points regarding the research and practice of Android development.

5.1. Performance-related smells remain a strong focus

As shown in the investigation of RQ1, research on Android-specific code smells is widely focused on uncovering performance issues. This suggests that performance is a major concern for both the end-users and the Android community. Several studies have shown how Android APIs can be misused (Khan et al., 2021b) and how developers can still maintain the coding style in traditional Java software without being aware of the performance sensitivity of Android devices.

On the one hand, Android devices have limited hardware resources to run apps. This means that an Android app is performance-sensitive software. The trivial practices in the OO program can easily cause battery drains and loading latency on this platform. On the other hand, with the rapid development of artificial intelligence techniques, various features (e.g., face recognition) have also been integrated into mobile apps. Although such features are convenient for end-users, developers neglect the performance, and computational cost of running machine learning models and communicating with servers in mobile systems (Ignatov et al., 2018).

Unfortunately, state-of-the-art approaches are ineffective for detecting performance issues using advanced techniques. With the evolution of Android version, it indicates that the research community still needs to explore new performance smells and effective detection and refactoring techniques to assist developers in optimizing the performance of Android apps. In addition, Android developers need to have more training on performance-oriented design to know the differences of computation capacity between OO programs and Android apps. For eliminating performance issues, developers may attempt to leverage various detection tools, such as AndroidLint.

5.2. Lack of consensus on Android-specific code smells

Reaching a consensus regarding the definitions of Android smells is important, as well as on how to measure the energy costs of code patterns. As shown in Table 6, researchers have also renamed existing smells based on their understanding. Such inconsistencies may cause researchers to diverge on the symptoms required to identify code smells (Kessentini and Ouni, 2017). In addition, some practitioners do not recognize the necessity of refactoring Android smells since they still doubt the negative impacts of smells on Android apps (Habchi et al., 2021b). To solve this issue, the research community needs to conduct more empirical studies on Android code smells from commercial apps to justify the impact of Android smells and the necessity of refactoring. In this case, practitioners can play a crucial role in providing significant data that researchers lack.

5.3. Holistic analysis for Android smells

As shown in Fig. 7, the median number of studied smells from the primary studies is much less than the total number of defined Android smells. We found that only DAAP (Oliveira et al., 2018) can identify 25 Android smells, whereas others can only detect up to 15. Therefore, developing analyzers that are capable of addressing more smells is necessary. As shown in Table 9, although detecting Android smells has received considerable attention from researchers, refactoring and prioritization are essential activities to assist practitioners in effectively addressing code smells with automated repair programs. Unfortunately, refactoring and prioritization have not been thoroughly investigated by the community. The examined studies are often built on top of other refactoring tools that address OO smells (Hamdi et al., 2021).

The research community should provide dedicated solutions for performing holistic analyses and optimizing the quality of Android apps. Since the end-users can perceive most Android-specific smells owing to battery drain and run-time performance, the user feedback can be considered to investigate smell prioritization. In addition, multiple studies have confirmed that performance-related smells may be harmless when isolated, whereas they can be particularly concerning when they occur in combination (Das et al., 2020; Anwar et al., 2019; Couto et al., 2020). Thus, the research community should consider such combined smells to optimize app quality when designing solutions for refactoring Android smells.

5.4. Considering various programming languages is essential for sound analysis

Three types of programming languages are available in Android development: Java, Kotlin, and JavaScript. The majority of studies reviewed in this SLR built their approaches based on various support tools for Java, such as Soot and WALA. In addition, Kotlin has received more attention from Android development practitioners because of its productivity and conciseness. With such popularity of Koltin, only one study analyzed Kotlin source code, as shown in Table 10. Although Kotlin is designed to interoperate fully with Java, existing tools for analyzing Java code cannot directly analyze Kotlin code because of the different syntax.

Furthermore, practitioners leverage cross-platform development techniques (e.g., Ionic) to build WebView apps for different mobile systems written in JavaScript. In this case, Java code is unavailable for analysis, even in the decompiled code. Unfortunately, we observed that the research community did not consider cross-platform development in any primary study, which was also revealed by Chan-Jong-Chu et al. (2020).

Overall, the current studies reviewed in this SLR do not consider Kotlin and JavaScript code in their implementations, missing the opportunity to uncover Android smells, leading to incomplete results. To solve this challenge, the research community should analyze other programming languages written in Android apps and provide relevant tools for addressing Android smells with various development frameworks.

5.5. Building a public benchmark

As shown in Table 14, most primary studies evaluated their approaches on a small set of open-source Android apps from F-Droid and GitHub, which may cause (1) such a dataset scale may not reflect the actual life cycle of Android smells; (2) researchers evaluated their results according to a golden set that was manually identified, which may exist bias owing to their different experiences. In addition, some studies applied machine-learning techniques to detect Android smells, which highly rely on the quality of the dataset.

Thus, we appeal that Android community should provide an open-source dataset with recognized instances of Android smells as a benchmark, as is combining empirical studies to investigate the acceptance of results among practitioners.

5.6. Combining multiple approaches could be the key towards a highly precise analysis

Current studies heavily depend on the API and software metrics used to detect code smells, as shown in Table 13. These features can be changed owing to the evolution of the Android framework, which may cause false negatives in detecting Android-specific smells (Khan et al., 2021b). For example, Prohibited Data Transfer (PDT) smell can be identified if `getBackgroundDataSetting` has not been checked before data transmission. After API 15, developers should apply `getNetworkCapabilities` to check whether the device is ready for transmission instead of it due to the deprecated API. The current detection techniques cannot address such issues of API changes. Therefore, the research community may consider combining natural language processing techniques, such as semantic similarity and code summarization, to analyze code semantics for detecting such API changes using similar semantics in both code and documentation levels. For instance, ACS-TNN (Yu et al., 2021) applies a tree-based neural network for Android code smells detection, retaining maximum semantic and structural information for extracting features of source codes.

In addition, existing approaches manually set threshold values for detecting smells, impacting the accuracy of metrics-based methods. Setting thresholds for Android smells still lacks consensus (Rasool and Ali, 2020). The research community should combine the practitioners' experience and explainable machine-learning techniques to automatically infer thresholds and association rules for a highly precise analysis.

6. Threats to validity

The results of an SLR may be subject to validity threats, mainly concerning the completeness and correctness of the SLR. In this section, we will summarize some implications for researchers and practitioners working in the Android smell domain. As proposed by Wohlin (2014), we have organized this section, including construct, internal, external, and conclusion validity threats.

6.1. Construct validity

Construct validity is related to the generalization of the result to the concept behind the study execution. In our case, it is related to the potentially subjective analysis of the selected publications. As recommended by Kitchenham's guidelines (Kitchenham, 2007), two authors performed data extraction independently. In case of discrepancies, a third author was involved in the discussion to clear up any disagreement. In addition, the quality of selected primary studies has been manually checked based on published venues and citations.

6.2. Internal validity

Internal validity threats are related to possible wrong findings about causal relationships between treatment and outcome. To address this threat, we carefully followed the guideline proposed by Kitchenham (2007). In this SLR, all findings were cross-checked among all authors. In case of disagreements, we discussed until a consensus was reached to guarantee the correct findings.

6.3. External validity

External validity threat is the ability to generalize the result (Wohlin, 2014). As the SLR, external validity relies on the validity of the selected publications. If the selected papers are not externally valid, the synthesis of their content will not be valid either. In our SLR, we were not able to evaluate the external validity of all selected studies.

6.4. Conclusion validity

Conclusion validity is related to the reliability of the conclusions drawn from the results (Wohlin, 2014). In our work, threats are related to the potential non-inclusion of some literature.

Although we attempted to collect as many relevant papers as possible from various digital libraries, our results may have missed some publications. To avoid this threat, we applied a board search string to collect all potential publications. In addition, we manually checked several relevant top-ranked conferences for missed publications, ensuring that influential publications were considered in our SLR as much as possible. To further mitigate this threat, we carefully applied the snowballing process (Wohlin, 2014), considering all references listed in the selected papers. In particular, we leveraged ConnectedPapers to discover potential papers according to the citation graph, resulting in two additional relevant papers. We defined the inclusion and exclusion criteria and first applied them to the title and abstract. However, we could not exclusively depend on these to establish whether the work reported evidence of Android-specific code smells. Before accepting the primary studies, all publications were cross-checked between all authors according to our selection criteria.

7. Related works

Given the large extension of either bad smells or Android analysis, as evidenced by the high number of studies conducted over decades, some surveys/SLRs have already been presented. Although they may have a few publications that we also studied, some well-known publications are missing owing to different goals. Indeed, our SLR has shown better coverage than others in terms of publications in the area of Android-specific smells.

Sobrinho et al. (2021) conducted an extensive literature review on code smells from 1990 to 2017, providing the concept of code smells and presenting current research trends in general code smells. This SLR discusses traditional OO smells and pinpoints new code smells in different contexts (e.g., Android smells and domain-specific languages). Their discussion also suggested that such Android smells would reveal new findings and patterns with practical outcomes.

Rasool and Ali (2020) presented a survey focusing on code smells in Android apps for both OO and Android smells, with the aim of identifying detection techniques and tools. They also proposed a prototype to detect Android-specific smells but limited the analysis of Java.

In an extreme sense, the existing literature reviews could not provide a thorough analysis of the various aspects of Android-specific code smells, such as affected quality, study method, and open challenges. However, our SLR focuses exclusively on Android-specific code smells, which differ from all SLRs mentioned. This SLR provides a better view of the landscape of Android-specific code smells to the Android community and discloses the current lack in this domain.

8. Conclusions

Software developers need to manage and refactor Android-specific smells since their presence is inevitable owing to various factors that relate to the evolution of the development environment and bad practices. Moreover, Android-specific smells that are mainly performance-oriented can be perceived by end-users more than traditional OO smells. Therefore, building a thorough overview of Android-specific code smells is necessary for the affected qualities, studied methods, and future directions.

Table A.1
Full list of primary studies.

No.	Year	Venue	Title
S01	2017	SANER	Lightweight Detection of Android-specific Code Smells: The aDoctor Project (Palomba et al., 2017)
S02	2017	MOBILESoft	P-lint: A permission smell detector for android applications (Dennis et al., 2017)
S03	2015	ICSE	An Approach to Detect Android Antipatterns (Hecht, 2015)
S04	2021	SAC	Exploiting the Progress of OO Refactoring Tools with Android Code Smells (Gattal et al., 2021)
S05	2019	MOBILESoft	Sniffing Android Code Smells: An Association Rules Mining-based Approach (Rubin et al., 2019)
S06	2019	MSR	The Rise of Android Code Smells: Who is to Blame? (Habchi et al., 2019a)
S07	2019	REDSET	Android Smells Detection using ML Algorithms with Static Code Metrics (Gupta et al., 2019)
S08	2018	EMSE	Security Code Smells in Android ICC (Gadient et al., 2019)
S09	2015	ASE	Study and Refactoring of Android Asynchronous Programming (Lin et al., 2015)
S10	2020	ICPC	Refactoring Android-specific Energy Smells: A Plugin for Android Studio (Iannone et al., 2020)
S11	2019	JSERD	Improving Energy Efficiency Through Automatic Refactoring (Cruz and Abreu, 2019)
S12	2016	MOBILESoft	Automated Re-factoring of Android Apps to Enhance Energy-efficiency (Banerjee and Roychoudhury, 2016)
S13	2017	MOBILESoft	Detecting Android Smells Using Multi-Objective Genetic Programming (Kessentini and Ouni, 2017)
S14	2020	TASE	Droidlens: Robust and Fine-Grained Detection for Android Code Smells (Mao et al., 2020)
S15	2019	EMSE	An Empirical catalog of code smells for the presentation layer of Android Apps (Carvalho et al., 2019)
S16	2018	TSE	EARMO: An Energy-Aware Refactoring Approach for Mobile Apps (Morales et al., 2017)
S17	2020	ASEW	Enforcing Green Code with Android Lint (Goaër, 2020)
S18	2020	SANER	Energy Refactorings for Android in the Large and in the Wild (Couto et al., 2020)
S19	2017	SANER	Investigating the Energy Impact of Android Smells (Carette et al., 2017)
S20	2020	ASEW	E-Debitum: Managing Software Energy Debt (Maia et al., 2020)
S21	2020	AJSE	Recovering Android Bad Smells from Android Applications (Rasool and Ali, 2020)
S22	2020	SANER	Web APIs in Android through the Lens of Security (Gadient et al., 2020)
S23	2017	MOBILESoft	M-Perm: A Lightweight Detector for Android Permission gaps (Chester et al., 2017)
S24	2020	QuASoQ	Detection and Correction of Android-specific Code Smells and Energy Bugs: An Android Lint Extension (Fatima et al., 2020)
S25	2016	Internetware	E-greenDroid: effective energy inefficiency analysis for android applications (Wang et al., 2016)
S26	2014	TSE	GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications (Liu et al., 2014b)
S27	2017	Internetware	NavyDroid: Detecting Energy Inefficiency Problems for Smartphone Applications (Liu et al., 2017)

(*continued on next page*)

This paper provides a systematic literature review to investigate the current relevant body of knowledge in Android-specific code smells and understand which smells have received more attention and what fundamental approaches have been proposed. Our SLR considers 35 primary studies published in software engineering and mobile-related conferences and journals, including

Table A.1 (*continued*).

S28	2018	SCP	State-taint analysis for detecting resource bugs (Xu et al., 2018)
S29	2014	ICSE	Characterizing and Detecting Performance Bugs for Smartphone Applications (Liu et al., 2014a)
S30	2018	ISSTA	Remove RATs from Your Code: Automated Optimization of Resource Inefficient Database Writes for Mobile Applications (Lyu et al., 2018)
S31	2014	FSE	Retrofitting concurrency for Android applications through refactoring (Lin et al., 2014)
S32	2018	TSE	EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps (Banerjee et al., 2017)
S33	2021	Electronics	Wake Lock Leak Detection in Android Apps Using Multi-Layer Perceptron (Khan et al., 2021b)
S34	2021	ICSE	Do not Do That! Hunting Down Visual Design Smells in Complex UIs against Design Guidelines (Yang et al., 2021)
S35	2021	QRS	A Novel Tree-based Neural Network for Android Code Smells Detection (Yu et al., 2021)

data on smells, affected quality, research activities, proposed methods, and evaluation strategies to address Android-specific smells.

The results of this SLR show that performance-related smells are by far the most frequently investigated since these smells generally cause memory and energy inefficiency in conjunction. The detection of Android smells in our examined studies can be performed using different approaches, all having different goals and proposing optimization with regard to different criteria without consensus from the research community. Building a solid, validated, and widely used set of datasets is necessary, in particular, as a benchmark for detecting Android smells. Moreover, the approaches proposed in most papers apply only to a small portion of known smells.

Currently, the Android source code written in Kotlin cannot be treated effectively owing to a lack of support tools for analysis. To analyze smells in Kotlin, primary studies converted Kotlin to Java via reverse engineering. Moreover, we found that the emerging cross-platform development of Android apps has not yet been considered in existing studies. With such development techniques, apps are generally written in JavaScript, which existing approaches cannot analyze.

Overall, this SLR provides an overview of Android-specific smells to the Android community. It contributes to practitioners and researchers in understanding the current strategies and challenges of Android smells. Since research on Android-specific code smells is still immature, the quality change before and after refactoring needs to be investigated to provide more evidence to the research community. In this case, more effective methods could be developed to refactor or prioritize Android-specific smells to overcome various challenges.

CRediT authorship contribution statement

Zhiqiang Wu: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft. **Xin Chen:** Validation, Formal analysis, Investigation, Resources, Data curation, Writing – review & editing, Visualization. **Scott Uk-Jin Lee:** Validation, Formal analysis, Investigation, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Data availability

We have shared figshare link in the paper to access our replication package.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-00155885, Artificial Intelligence Convergence Innovation Human Resources Development (Hanyang University ERICA)).

Appendix. Full list of primary studies

See Table A.1.

References

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2016. Androzoo: Collecting millions of android apps for the research community. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories. MSR, IEEE, pp. 468–471. http://dx.doi.org/10.1145/2901739.2903508.

Android, 2012. Accessibility: Are you serving all your users?. URL https://android-developers.googleblog.com/2012/04/accessibility-are-you-serving-all-your.html.

Android, 2017. Android lint checks. URL https://sites.google.com/a/android.com/tools/tips/lint-checks.

Anon, 2022a. Market shares of Kotlin in top apps. https://www.appbrain.com/stats/libraries/details/kotlin/kotlin. (Accessed 25 May 2022).

Anon, 2022b. Mobile app development: Market share, size, and other statistics. https://www.appmysite.com/blog/mobile-app-development-market-share. (Accessed 10 February 2022).

Anwar, H., 2020. Towards greener android application development. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings. ICSE-Companion, IEEE, pp. 170–173. http://dx.doi.org/10.1145/3377812.3381390.

Anwar, H., Pfahl, D., Srirama, S.N., 2019. Evaluating the impact of code smell refactoring on the energy consumption of android applications. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 82–86. http://dx.doi.org/10.1109/SEAA.2019.00021.

Ardito, L., Coppola, R., Malnati, G., Torchiano, M., 2020. Effectiveness of Kotlin vs. Java in android app development tasks. Inf. Softw. Technol. 127, 106374. http://dx.doi.org/10.1016/j.infsof.2020.106374.

Banerjee, A., Chong, L.K., Ballabriga, C., Roychoudhury, A., 2017. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. IEEE Trans. Softw. Eng. 44 (5), 470–490.

Banerjee, A., Chong, L.K., Chattopadhyay, S., Roychoudhury, A., 2014. Detecting energy bugs and hotspots in mobile apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 588–598.

Banerjee, A., Roychoudhury, A., 2016. Automated re-factoring of android apps to enhance energy-efficiency. In: Proceedings of the International Conference on Mobile Software Engineering and Systems. pp. 139–150. http://dx.doi.org/10.1145/2897073.2897086.

Bartel, A., Klein, J., Le Traon, Y., Monperrus, M., 2012. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. pp. 27–38. http://dx.doi.org/10.1145/2259051.2259056.

Bavota, G., Lucia, A.D., Penta, M.D., Oliveto, R., Palomba, F., 2015. An experimental investigation on the innate relationship between quality and refactoring. J. Syst. Softw. 107, http://dx.doi.org/10.1016/j.jss.2015.05.024.

Boutaib, S., Bechikh, S., Palomba, F., Elarbi, M., Makhlouf, M., Said, L.B., 2021. Code smell detection and identification in imbalanced environments. Expert Syst. Appl. 166, http://dx.doi.org/10.1016/j.eswa.2020.114076.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al., 2010. Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. pp. 47–52. http://dx.doi.org/10.1145/1882362.1882373.

Campbell, G.A., Papapetrou, P.P., 2013. SonarQube in Action. Manning Publications Co.

Carette, A., Younes, M.A.A., Hecht, G., Moha, N., Rouvoy, R., 2017. Investigating the energy impact of android smells. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 115–126.

Carvalho, S.G., Aniche, M., Veríssimo, J., Durelli, R.S., Gerosa, M.A., 2019. An empirical catalog of code smells for the presentation layer of android apps. Empir. Softw. Eng. 24 (6), 3546–3586.

Chan-Jong-Chu, K., Islam, T., Exposito, M.M., Sheombar, S., Valladares, C., Philippot, O., Grua, E.M., Malavolta, I., 2020. Investigating the correlation between performance scores and energy consumption of mobile web apps. In: Proceedings of the Evaluation and Assessment in Software Engineering. pp. 190–199. http://dx.doi.org/10.1145/3383219.3383239.

Chen, J., Chen, C., Xing, Z., Xu, X., Zhu, L., Li, G., Wang, J., 2020. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering. ICSE, IEEE, pp. 322–334.

Chester, P., Jones, C., Mkaouer, M.W., Krutz, D.E., 2017. M-perm: A lightweight detector for android permission gaps. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 217–218. http://dx.doi.org/10.1109/MOBILESoft.2017.23.

Couto, M., Saraiva, J., Fernandes, J.P., 2020. Energy refactorings for android in the large and in the wild. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 217–228. http://dx.doi.org/10.1109/SANER48275.2020.9054858.

Cruz, L., Abreu, R., 2017. Performance-based guidelines for energy efficient mobile applications. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 46–57.

Cruz, L., Abreu, R., 2019. Improving energy efficiency through automatic refactoring. J. Softw. Eng. Res. Dev. 7, http://dx.doi.org/10.5753/jserd.2019.17.

Cruz, L., Abreu, R., Rouvignac, J.-N., 2017. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 205–206. http://dx.doi.org/10.1109/MOBILESoft.2017.21.

Das, T., Di Penta, M., Malavolta, I., 2016. A quantitative and qualitative investigation of performance-related commits in android apps. In: 2016 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 443–447. http://dx.doi.org/10.1109/ICSME.2016.49.

Das, T., Di Penta, M., Malavolta, I., 2020. Characterizing the evolution of statically-detectable performance issues of android apps. Empir. Softw. Eng. 25 (4), 2748–2808. http://dx.doi.org/10.1007/s10664-019-09798-3.

De Stefano, M., Gambardella, M.S., Pecorelli, F., Palomba, F., De Lucia, A., 2020. cASpER: A plug-in for automated code smell detection and refactoring. In: Proceedings of the International Conference on Advanced Visual Interfaces. pp. 1–3. http://dx.doi.org/10.1145/3399715.3399955.

Dennis, C., Krutz, D.E., Mkaouer, M.W., 2017. P-lint: A permission smell detector for android applications. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 219–220. http://dx.doi.org/10.1109/MOBILESoft.2017.24.

Desnos, A., 2012. Android: Static analysis using similarity distance. In: 2012 45th Hawaii International Conference on System Sciences. IEEE, pp. 5394–5403. http://dx.doi.org/10.1109/HICSS.2012.114.

Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., De Lucia, A., 2017a. Petra: a software-based tool for estimating the energy profile of android applications. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion. ICSE-C, IEEE, pp. 3–6. http://dx.doi.org/10.1109/ICSE-C.2017.18.

Di Nucci, D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., De Lucia, A., 2017b. Software-based energy profiling of android apps: Simple, efficient and reliable? In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 103–114. http://dx.doi.org/10.1109/SANER.2017.7884613.

Fang, C., Liu, Z., Shi, Y., Huang, J., Shi, Q., 2020. Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 516–527. http://dx.doi.org/10.1145/3395363.3397362.

Fatima, I., Anwar, H., Pfahl, D., Qamar, U., 2020. Detection and correction of android-specific code smells and energy bugs: An android lint extension. In: QuASoQ@ APSEC. pp. 71–78.

Flauzino, M., Veríssimo, J., Terra, R., Cirilo, E., Durelli, V.H., Durelli, R.S., 2018. Are you still smelling it? A comparative study between Java and Kotlin language. In: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse. pp. 23–32. http://dx.doi.org/10.1145/3267183.3267186.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Berkeley, CA, USA.

Gadient, P., Ghafari, M., Frischknecht, P., Nierstrasz, O., 2019. Security code smells in android ICC. Empir. Softw. Eng. 24 (5), 3046–3076. http://dx.doi.org/10.1007/s10664-018-9673-y.

Gadient, P., Ghafari, M., Tarnutzer, M.-A., Nierstrasz, O., 2020. Web apis in android through the lens of security. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 13–22. http://dx.doi.org/10.1109/SANER48275.2020.9054850.

Gao, Y., Wang, Z., Liu, S., Yang, L., Sang, W., Cai, Y., 2019. Teccd: A tree embedding approach for code clone detection. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 145–156. http://dx.doi.org/10.1109/ICSME.2019.00025.

Gattal, A., Hammache, A., Bousbia, N., Henniche, A.N., 2021. Exploiting the progress of OO refactoring tools with Android code smells: RAndroid, a plugin for Android studio. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1580–1583. http://dx.doi.org/10.1145/3412841.3442129.

Ghafari, M., Gadient, P., Nierstrasz, O., 2017. Security smells in android. In: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 121–130.

Ghari, S., Hadian, M., Rasolroveicy, M., Fokaefs, M., 2019. A multi-dimensional quality analysis of android applications. In: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering. pp. 34–43.

Goaër, O.L., 2020. Enforcing green code with android lint. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops. pp. 85–90. http://dx.doi.org/10.1145/3417113.3422188.

Góis Mateus, B., Martinez, M., 2019. An empirical study on quality of Android applications written in Kotlin language. Empir. Softw. Eng. 24 (6), 3356–3393.

Grano, G., Di Sorbo, A., Mercaldo, F., Visaggio, C.A., Canfora, G., Panichella, S., 2017. Android apps and user feedback: a dataset for software evolution and quality improvement. In: Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics. pp. 8–11.

Gupta, A., Suri, B., Bhat, V., 2019. Android smells detection using ML algorithms with static code metrics. In: International Conference on Recent Developments in Science, Engineering and Technology. Springer, pp. 64–79.

Habchi, S., 2019. Understanding Mobile-Specific Code Smells (Ph.D. thesis). Université de Lille.

Habchi, S., Moha, N., Rouvoy, R., 2019a. The rise of android code smells: Who is to blame? In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 445–456. http://dx.doi.org/10.1109/MSR.2019.00071.

Habchi, S., Moha, N., Rouvoy, R., 2021b. Android code smells: From introduction to refactoring. J. Syst. Softw. 177, http://dx.doi.org/10.1016/j.jss.2021.110964.

Habchi, S., Rouvoy, R., Moha, N., 2019. On the survival of android code smells in the wild. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 87–98. http://dx.doi.org/10.1109/MOBILESoft.2019.00022.

Hamdi, O., Ouni, A., Cinnéide, M.Ó., Mkaouer, M.W., 2021. A longitudinal study of the impact of refactoring in android applications. Inf. Softw. Technol. 140, 106699. http://dx.doi.org/10.1016/j.infsof.2021.106699.

Hecht, G., 2015. An approach to detect android antipatterns. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, pp. 766–768. http://dx.doi.org/10.1109/ICSE.2015.243.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N., Duchien, L., 2015a. Tracking the software quality of android applications along their evolution (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 236–247. http://dx.doi.org/10.1109/ASE.2015.46.

Hecht, G., Moha, N., Rouvoy, R., 2016. An empirical study of the performance impacts of android code smells. In: Proceedings of the International Conference on Mobile Software Engineering and Systems. pp. 59–69. http://dx.doi.org/10.1145/2897073.2897100.

Hecht, G., Rouvoy, R., Moha, N., Duchien, L., 2015b. Detecting antipatterns in android apps. In: 2015 2nd ACM International Conference on Mobile Software Engineering and Systems. IEEE, pp. 148–149. http://dx.doi.org/10.1109/MobileSoft.2015.38.

Iannone, E., Pecorelli, F., Di Nucci, D., Palomba, F., De Lucia, A., 2020. Refactoring android-specific energy smells: A plugin for android studio. In: Proceedings of the 28th International Conference on Program Comprehension. pp. 451–455. http://dx.doi.org/10.1145/3387904.3389298.

Ignatov, A., Timofte, R., Chou, W., Wang, M.W.K., Hartley, T., Gool, L.V., 2018. AI benchmark: Running deep neural networks on android smartphones. In: Proceedings of the European Conference on Computer Vision, Vol. 11133. ECCV.

Jiang, D., Ma, P., Su, X., Wang, T., 2014. Distance metric based divergent change bad smell detection and refactoring scheme analysis. Int. J. Innovative Comput. Inf. Control 10.

Kaur, S., Singh, P., 2019. How does object-oriented code refactoring influence software quality? Research landscape and challenges. J. Syst. Softw. 157, http://dx.doi.org/10.1016/j.jss.2019.110394.

Kessentini, M., Ouni, A., 2017. Detecting android smells using multi-objective genetic programming. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 122–132. http://dx.doi.org/10.1109/MOBILESoft.2017.29.

Khan, M.U., Abbas, S., Lee, S.U.-J., Abbas, A., 2021a. Measuring power consumption in mobile devices for energy sustainable app development: A comparative study and challenges. Sustain. Comput.: Inform. Syst. 31, 100589.

Khan, M.U., Lee, S.U.-J., Wu, Z., Abbas, S., 2021b. Wake lock leak detection in android apps using multi-layer perceptron. Electronics 10 (18), 2211. http://dx.doi.org/10.3390/electronics10182211.

Khomh, F., Di Penta, M., Gueheneuc, Y.-G., 2009. An exploratory study of the impact of code smells on software change-proneness. In: 2009 16th Working Conference on Reverse Engineering. IEEE, pp. 75–84. http://dx.doi.org/10.1109/WCRE.2009.28.

Kitchenham, B., 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report, Ver. 2.3 EBSE Technical Report, EBSE.

Kuutila, M., Mantyla, M.V., Farooq, U., Claes, M., 2021. What do we know about time pressure in software development? IEEE Softw. 38, http://dx.doi.org/10.1109/MS.2020.3020784.

Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.G., 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. J. Syst. Softw. 167, http://dx.doi.org/10.1016/j.jss.2020.110610.

Lam, P., Bodden, E., Lhoták, O., Hendren, L., 2011. The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop. CETUS 2011, pp. 1–43.

Lei, M., Li, H., Li, J., Aundhkar, N., Kim, D.-K., 2022. Deep learning application on code clone detection: A review of current knowledge. J. Syst. Softw. 184, 111141. http://dx.doi.org/10.1016/j.jss.2021.111141.

Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, L., 2017. Static analysis of android apps: A systematic literature review. Inf. Softw. Technol. 88, 67–95. http://dx.doi.org/10.1016/j.infsof.2017.04.001.

Li, D., Halfond, W.G., 2014. An investigation into energy-saving programming practices for android smartphone app development. In: Proceedings of the 3rd International Workshop on Green and Sustainable Software. pp. 46–53.

Lin, Y., Okur, S., Dig, D., 2015. Study and refactoring of android asynchronous programming (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 224–235. http://dx.doi.org/10.1109/ASE.2015.50.

Lin, Y., Radoi, C., Dig, D., 2014. Retrofitting concurrency for android applications through refactoring. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 341–352. http://dx.doi.org/10.1145/2635868.2635903.

Liu, Y., Wang, J., Xu, C., Ma, X., 2017. NavyDroid: detecting energy inefficiency problems for smartphone applications. In: Proceedings of the 9th Asia-Pacific Symposium on Internetware. pp. 1–10. http://dx.doi.org/10.1145/3131704.3131705.

Liu, Y., Xu, C., Cheung, S.-C., 2014a. Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 36th International Conference on Software Engineering. pp. 1013–1024. http://dx.doi.org/10.1145/2568225.2568229.

Liu, Y., Xu, C., Cheung, S.-C., Lü, J., 2014b. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. IEEE Trans. Softw. Eng. 40 (9), 911–940. http://dx.doi.org/10.1109/TSE.2014.2323982.

Lyu, Y., Alotaibi, A., Halfond, W.G., 2019. Quantifying the performance impact of SQL antipatterns on mobile applications. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 53–64. http://dx.doi.org/10.1109/ICSME.2019.00015.

Lyu, Y., Li, D., Halfond, W.G., 2018. Remove rats from your code: automated optimization of resource inefficient database writes for mobile applications. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 310–321. http://dx.doi.org/10.1145/3213846.3213865.

Maia, D., Couto, M., Saraiva, J., Pereira, R., 2020. E-debitum: managing software energy debt. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops. ASEW, IEEE, pp. 170–177. http://dx.doi.org/10.1145/3417113.3422999.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y.-G., Aimeur, E., 2012. Smurf: A svm-based incremental anti-pattern detection approach. In: 2012 19th Working Conference on Reverse Engineering. IEEE, pp. 466–475. http://dx.doi.org/10.1109/WCRE.2012.56.

Mao, C., Wang, H., Han, G., Zhang, X., 2020. Droidlens: Robust and fine-grained detection for android code smells. In: 2020 International Symposium on Theoretical Aspects of Software Engineering. TASE, IEEE, pp. 161–168. http://dx.doi.org/10.1109/TASE49443.2020.00030.

Marimuthu, C., Chandrasekaran, K., Chimalakonda, S., 2021. Energy diagnosis of android applications: A thematic taxonomy and survey. ACM Comput. Surv. 53, 1–36. http://dx.doi.org/10.1145/3417986.

Marinescu, R., 2004. Detection strategies: Metrics-based rules for detecting design flaws. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. IEEE, pp. 350–359. http://dx.doi.org/10.1109/ICSM.2004.1357820.

Martin, R.C., Grenning, J., Brown, S., Henney, K., Gorman, J., 2018. Clean Architecture: A Craftsman's Guide to Software Structure and Design, Vol. 31. Prentice Hall.

Martins, J., Bezerra, C., Uchôa, A., Garcia, A., 2021. How do code smell co-occurrences removal impact internal quality attributes? A developers' perspective. In: Brazilian Symposium on Software Engineering. pp. 54–63.

Mazuera-Rozo, A., Trubiani, C., Linares-Vásquez, M., Bavota, G., 2020. Investigating types and survivability of performance bugs in mobile apps. Empir. Softw. Eng. 25, http://dx.doi.org/10.1007/s10664-019-09795-6.

Morales, R., Chicano, F., Khomh, F., Antoniol, G., 2018. Efficient refactoring scheduling based on partial order reduction. J. Syst. Softw. 145, http://dx.doi.org/10.1016/j.jss.2018.07.076.

Morales, R., Saborido, R., Khomh, F., Chicano, F., Antoniol, G., 2017. Earmo: An energy-aware refactoring approach for mobile apps. IEEE Trans. Softw. Eng. 44 (12), 1176–1206. http://dx.doi.org/10.1109/TSE.2017.2757486.

Oliveira, V., Teixeira, L., Ebert, F., 2020. On the adoption of Kotlin on android development: A triangulation study. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 206–216. http://dx.doi.org/10.1109/SANER48275.2020.9054859.

Oliveira, J., Viggiato, M., Santos, M.F., Figueiredo, E., Marques-Neto, H., 2018. An empirical study on the impact of android code smells on resource usage.. In: SEKE. pp. 313–314. http://dx.doi.org/10.18293/SEKE2018-157.

Opdyke, W.F., 1992. Refactoring Object-Oriented Frameworks. Citeseer.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Hamdi, M.S., 2015. Improving multi-objective code-smells correction using development history. J. Syst. Softw. 105, http://dx.doi.org/10.1016/j.jss.2015.03.040.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2013. Detecting bad smells in source code using change history information. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 268–278. http://dx.doi.org/10.1109/ASE.2013.6693086.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., De Lucia, A., 2014. Mining version histories for detecting code smells. IEEE Trans. Softw. Eng. 41 (5), 462–489. http://dx.doi.org/10.1109/TSE.2014.2372760.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A., 2017. Lightweight detection of android-specific code smells: The adoctor project. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 487–491. http://dx.doi.org/10.1109/SANER.2017.7884659.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A., 2019. On the impact of code smells on the energy consumption of mobile applications. Inf. Softw. Technol. 105, http://dx.doi.org/10.1016/j.infsof.2018.08.004.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L., 2016. Spoon: A library for implementing analyses and transformations of java source code. Softw. - Pract. Exp. 46 (9), 1155–1179. http://dx.doi.org/10.1002/spe.2346.

Peruma, A., 2019. A preliminary study of android refactorings. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 148–149. http://dx.doi.org/10.1109/MOBILESoft.2019.00030.

Prestat, D., Moha, N., Villemaire, R., 2022. An empirical study of android behavioural code smells detection. Empir. Softw. Eng. 27 (7), 1–34.

Rahman, A., Parnin, C., Williams, L., 2019. The seven sins: Security smells in infrastructure as code scripts. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 164–175.

Rahman, A., Rahman, M.R., Parnin, C., Williams, L., 2021. Security smells in ansible and chef scripts: A replication study. ACM Trans. Softw. Eng. Methodol. 30 (1), 1–31.

Rahman, A., Williams, L., 2021. Different kind of smells: Security smells in infrastructure as code scripts. IEEE Secur. Priv. 19 (3), 33–41.

Rasool, G., Ali, A., 2020. Recovering android bad smells from android applications. Arab. J. Sci. Eng. 45 (4), 3289–3315. http://dx.doi.org/10.1007/s13369-020-04365-1.

Reimann, J., 2014. A tool-supported quality smell catalogue for android developers. In: Proc. of the Conference ….
Reimann, J., Aßmann, U., 2013. Quality-aware refactoring for early detection and resolution of energy deficiencies. In: 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing. IEEE, pp. 321–326. http://dx.doi.org/10.1109/UCC.2013.70.
Rubin, J., Henniche, A.N., Moha, N., Bouguessa, M., Bousbia, N., 2019. Sniffing android code smells: an association rules mining-based approach. In: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems. MOBILESoft, IEEE, pp. 123–127. http://dx.doi.org/10.1109/MOBILESoft.2019.00025.
Salehie, M., Li, S., Tahvildari, L., 2006. A metric-based heuristic framework to detect object-oriented design flaws. In: 14th IEEE International Conference on Program Comprehension. ICPC'06, IEEE, pp. 159–168. http://dx.doi.org/10.1109/ICPC.2006.6.
Scoccia, G.L., Peruma, A., Pujols, V., Christians, B., Krutz, D., 2019. An empirical history of permission requests and mistakes in open source android apps. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 597–601. http://dx.doi.org/10.1109/MSR.2019.00090.
Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D., 2021. Code smell detection by deep direct-learning and transfer-learning. J. Syst. Softw. 176, http://dx.doi.org/10.1016/j.jss.2021.110936.
Sharma, T., Mishra, P., Tiwari, R., 2016. Designite: A software design quality assessment tool. In: Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities. pp. 1–4. http://dx.doi.org/10.1145/2896935.2896938.
Sharma, T., Spinellis, D., 2018. A survey on software smells. J. Syst. Softw. 138, http://dx.doi.org/10.1016/j.jss.2017.12.034.
Shoenberger, I., Mkaouer, M.W., Kessentini, M., 2017. On the use of smelly examples to detect code smells in JavaScript. In: European Conference on the Applications of Evolutionary Computation. Springer, pp. 20–34. http://dx.doi.org/10.1007/978-3-319-55792-2_2.
Silva, C.D.S., Ferreira da Costa, L., Rocha, L.S., Viana, G.V.R., 2020. KNN applied to PDG for source code similarity classification. In: Brazilian Conference on Intelligent Systems. Springer, pp. 471–482. http://dx.doi.org/10.1007/978-3-030-61380-8_32.
Sobrinho, E.V.D.P., Lucia, A.D., Maia, M.D.A., 2021. A systematic literature review on bad smells-5 W's: Which, when, what, who, where. IEEE Trans. Softw. Eng. 47, http://dx.doi.org/10.1109/TSE.2018.2880977.
Soh, Z., Yamashita, A., Khomh, F., Guéhéneuc, Y.-G., 2016. Do code smells impact the effort of different maintenance programming activities? In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 1. SANER, IEEE, pp. 393–402.
Suryanarayana, G., Samarthyam, G., Sharma, T., 2014. Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann, http://dx.doi.org/10.1016/C2013-0-23413-9.
Visser, W., Păsăreanu, C.S., Khurshid, S., 2004. Test input generation with Java PathFinder. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 97–107. http://dx.doi.org/10.1145/1013886.1007526.
Wang, J., Liu, Y., Xu, C., Ma, X., Lu, J., 2016. E-greenDroid: effective energy inefficiency analysis for android applications. In: Proceedings of the 8th Asia-Pacific Symposium on Internetware. pp. 71–80. http://dx.doi.org/10.1145/2993717.2993720.
Wang, M., Wang, P., Xu, Y., 2017. CCSharp: An efficient three-phase code clone detector using modified PDGs. In: 2017 24th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 100–109.
Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–10.
Wu, Z., Chen, X., Khan, M.U., Lee, S.U.-J., 2021. Enhancing fidelity of description in android apps with category-based common permissions. IEEE Access 9, 105493–105505.
Wu, Z., Chen, X., Lee, S.U.-J., 2020. FCDP: Fidelity calculation for description-to-permissions in android apps. IEEE Access 9, 1062–1075.
Xu, Z., Wen, C., Qin, S., 2018. State-taint analysis for detecting resource bugs. Sci. Comput. Program. 162, 93–109. http://dx.doi.org/10.1016/j.scico.2017.06.010.
Yamashita, A., Moonen, L., 2013. To what extent can maintenance problems be predicted by code smell detection? -An empirical study. Inf. Softw. Technol. 55, http://dx.doi.org/10.1016/j.infsof.2013.08.002.
Yang, B., Xing, Z., Xia, X., Chen, C., Ye, D., Li, S., 2021. Don't do that! hunting down visual design smells in complex uis against design guidelines. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 761–772. http://dx.doi.org/10.1109/icse43902.2021.00075.
Yu, J., Mao, C., Ye, X., 2021. A novel tree-based neural network for android code smells detection. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 738–748.
Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 783–794. http://dx.doi.org/10.1109/ICSE.2019.00086.

Zhiqiang Wu is a security engineer at Honor Devices Co. Ltd. He received the B.S. in computer science from Shanghai Polytechnic University, China, in 2015. He also received M.S. and Ph.D. degrees in computer science from Hanyang University in 2017 and 2022, respectively. His research interests include mobile app analysis, code smells, and software refactoring on mobile apps.

Xin Chen was born in Liaoning, China, in 1995. She received her B.S. degrees in software engineering from the Harbin University of Science and Technology, China, in 2017. She is currently pursuing the M.S. degree leading to Ph.D. program with the Department of Computer Science and Engineering, Hanyang University, Ansan, South Korea. Her research interests include software smell detection, code refactoring, and code plagiarism.

Scott Uk-Jin Lee received his B.S. degree in software engineering and Ph.D. degree in computer science from the University of Auckland, New Zealand. After the Ph.D. degree, he worked as a Post-Doctoral Research Fellow at the Commissariat à l'énergieatomique et aux énergies alternatives, France. He is currently a Professor in the College of Computing at Hanyang University ERICA, South Korea. He is also the Director of Software Convergence Institute and Software Education Center at Hanyang University ERICA. His research interests include software engineering, formal methods, and quality assurance. He is also a member of the Korean Institute of Information Scientists and Engineers and the Korean Society of Computer and Information. He has served as an editor, technical chair, and committee member for several journals and conferences.