Chapter 7
Tool Support for Green Android Development

Hina Anwar, Iffat Fatima, Dietmar Pfahl, and Usman Qamar

Abstract Mobile applications are developed with limited battery resources in mind. To build energy-efficient mobile apps, many support tools have been developed which aid developers during the development and maintenance phases. To understand what is already available and what is still needed to support green Android development, we conducted a systematic mapping study to overview the state of the art and to identify further research opportunities. After applying inclusion/exclusion and quality criteria, we identified tools for detecting/refactoring code smells/energy bugs, and for detecting/migrating third-party libraries in Android applications. The main contributions of this study are: (1) classification of identified tools based on the support they offer to aid green Android development, (2) classification of the identified tools based on techniques used to offer support to developers, and (3) characterization of the identified tools based on the user interface, IDE integration, and availability. The most important finding is that the tools for detecting/migrating third-party libraries in Android development do not provide support to developers to optimize code w.r.t. energy consumption, which merits further research.

7.1 Introduction

Global warming due to CO_2 emissions has been one of the most prominent environmental issues in the past decade. A part of these CO_2 emissions is contributed by the information and communication technology (ICT) industry [1]. Therefore, producing green or sustainable products and practices has been the focus of many researchers in the ICT community. Recently, however, the focus of research in the

H. Anwar (✉) · D. Pfahl
Institute of Computer Science, University of Tartu, Tartu, Estonia
e-mail: hina.anwar@ut.ee; dietmar.pfahl@ut.ee

I. Fatima · U. Qamar
College of Electrical and Mechanical Engineering, National University of Sciences and Technology, Islamabad, Pakistan
e-mail: iffat.fatima@ce.ceme.edu.pk; usmanq@ceme.nust.edu.pk

C. Calero et al. (eds.), *Software Sustainability*,
https://doi.org/10.1007/978-3-030-69970-3_7

ICT community has shifted from optimizing the energy consumption of hardware to optimizing the energy consumption of software [2–8], as software indirectly consumes energy by controlling the equipment. An efficiently designed software might use resources optimally, thus reducing energy consumption [9–11]. Among portable devices, mobile phones are the most commonly used. Statistics show that the usage of mobile devices will grow in the coming years [12], indicating an increase in the carbon footprint.

Green software development encompasses green by software and green in software. Green by software means using software products to make other domains of life more sustainable. Green in software refers to the study and practice of designing, developing, maintaining, and disposing of software products in such a way that they have a minimal negative impact on the environment, community, economy, individuals, and technology [13, 14].

This chapter mostly focuses on green in software and summarizes the tool support available to improve the green-ability of Android apps in the development and maintenance phases. The term "Android development" refers to the development of applications that are developed to operate on devices running the Android operating system. These applications can be developed in various languages; however, in this chapter, we focus on Android development in Java. Android development differs from traditional software development in terms of context, user experience, and a touch-based interface. Android applications are designed for portable devices, which have limited resources such as memory or battery. A common struggle during Android application development is how to make the applications efficient in terms of resource usage. Banarjee et al. summarize the problem nicely as follows: "High computational power coupled with small battery capacity and the application development in an energy-oblivious fashion can only lead to one situation: short battery life and an unsatisfied user base" [15].

Previous studies have explored applications in app stores in order to define procedures to optimize their energy consumption [16–23]. Some studies have focused on profiling energy [24–28] consumed by applications, while others have developed support tools [29, 30]. As compared to desktop or web applications, Android applications contain multiple components that have user-driven workflows. A typical Android application consists of activities, fragments, services, content providers, and broadcast receivers. Due to the difference in architecture, the support tools used in the development of traditional Java-based applications are not so useful in Android application development and maintenance. Android application code can be roughly divided into two part: custom code and reusable code. While custom code is unique to each app, reusable code includes third-party libraries that are included in apps to speed up the development process.

In the domain of Android application development, research has been focused on development activities related to energy efficiency, memory usage, performance, etc., and maintenance activities related to code smell detection and correction, energy bug detection and correction, detection/migration of third-party libraries, etc.

Code smells are an indication of possible problems in source code or design of the applications. Such problems can be avoided by refactoring the code [31]. However,

object-oriented code smells are different from Android-specific code smells. In Android development, code smell can appear due to frequent development and update cycles of applications. Some studies [32–34] have focused on identifying, cataloguing, and profiling the energy consumption of Android-specific code smells. Energy bugs are scenarios which cause unexpected energy drains such as preventing the mobile device from going into the idle state even after the application execution has completed. Such malfunctioning can cause battery drain and should be avoided [15]. To build an energy-efficient Android application developers need to identify and refactor code smells/energy bugs.

Third-party libraries are reusable components available to implement various functionalities in the app, such as billing, advertisement, and networking. Up until June 2020, the online Maven repository[1] contained 344,869 unique libraries. Such a huge supply of third-party libraries is linked to the demands and needs of developers [35]. Almost 60% of code in Android applications is related to third-party libraries [36]. However, these libraries could introduce various security-, privacy-, permission-, and resource usage-related issues in applications [37]. The research on the detection/migration of third-party libraries has many uses. Some studies have used third-party library detection techniques for finding security vulnerabilities [38–41] in Android apps, while others have focused on privacy leaks [42–46]. Third-party libraries have been detected and removed as noise in clone, app repackage, and malicious app detection studies [47–51]. Third-party libraries are detected and removed from these studies in order to improve the accuracy of the analysis. Studies related to the energy impact of third-party Android libraries are limited [52].

In order to build effective Android-specific support tools to aid green Android development, we first need to understand what is already available, what is still needed, and how the problems in existing tools can be overcome. Based on published literature we outlined an explorative analysis of support tools available to (1) optimize code in Android applications through code smell detection/refactoring, and (2) optimize reusable code in Android applications through detection/migration of third-party libraries. This study extends our previous work [53] comparing 21 tools in the following ways: we have improved search string and extended our analysis for one more year, which gave us 30 more tools and also additional results. We provide further information about the interface, availability, and integrated development environment (IDE) integration of all 51 tools.

The remainder of this chapter is organized as follows. Section 7.2 presents related work. Section 7.3 describes the methodology used to analyze the literature. Section 7.4 presents the result of screening the publications and classification and analysis. Section 7.5 provides a discussion to identify future research directions. Section 7.6 provides possible threats to the validity of this study. Section 7.7 concludes the chapter by summarizing the main findings.

[1] https://search.maven.org/stats, statistics for central repository

7.2 Related Work

Secondary studies related to energy efficiency in Android development are scarce. Some [54–56] have reviewed tools and techniques for improving the quality of Java projects in the object-oriented paradigm (with regard to performance or maintainability).

Most Android projects use Java as the programing language; however, the support tools and techniques used for Java projects reviewed by previous secondary studies [54–56] cannot be effectively applied to Android projects. Therefore, many specialized support tools have been developed to improve the quality of Android apps with regard to maintainability, performance, security, or energy. Li et al. [57] performed a systematic literature review to analyze static source code analysis techniques and tools proposed for Android to assess issues related to security, performance, or energy. The authors have reviewed work published between 2011 and 2015, consisting of 124 studies. The review concluded that the majority of static analysis techniques only uncover security flaws in Android apps. Degu A [58]. performed a systematic literature review to classify primary studies with a focus on resource usage, energy consumption, and performance in Android apps. The classification is high level based on the main research focus, type of contribution, and type of evaluation method adopted in selected studies. Their results did not provide an in-depth review of support tools in green Android development.

Another group of studies has compared the state-of-the-art tools through experiments in order to benchmark their performance, accuracy, and reporting capabilities. Qiu et al. [59] provide a comparison between three static analysis tools: FlowDroid/IccTA, Amandroid, and Droidsafe. They evaluated these tools using a common configuration setup and the same set of benchmark applications. Results were compared to those of previous studies in order to identify reasons for inaccuracy in existing tools. Corrodi et al. [60] review the state-of-the-art in Android data leak detection tools. Out of 87 state-of-the-art tools, they executed five based on availability. They compared these five tools against a set of known vulnerabilities and discussed the overall performance of the tools. Ndagi and Alhassan [61] provide a comparison of machine classifiers for detecting phishing attacks in Adware in Android applications. This study concluded that many existing machine classifiers, if adequately explored, could yield more accurate results for phishing detection.

Another group of studies has focused on reviewing the technique related to security, malware, similarity, and repackaging in Android apps. Cooper et al. [62] provide an overview of security threats posed by Android malware. They also survey some common defense techniques to mitigate the impact of malware applications and characteristics that are commonly found in malware applications that could enable detection techniques. Li et al. [63] provide a literature review that summarizes the challenges and current solutions for detecting repackaged apps. They concluded that many existing solutions merit further research as they are tested on closed datasets and might not be as efficient or accurate as they claim to be. Roy et al.

[64] provide a qualitative comparison of clone detection techniques and tools. They classify, compare, and evaluate these tools.

We found some studies that have conducted controlled experiments to measure the energy consumption of third-party libraries. For example, Wang et al. [65] presented an algorithmic solution to model the energy minimization problem for ad prefetching in Android apps. Rasmussen et al. [66] conducted a study to compare the power efficiency of various methods of blocking advertisements on an Android platform. They found many cases where ad-blocking software or methods resulted in increased power usage. In Android applications, there could be many reasons for long-running operations in the background that continuously consume resources. Such operations could cause battery drain and performance degradation. Shao et al. [67] demonstrated through an experiment that sometimes such behavior is not intentional and is caused by third-party libraries.

However, we could not find any secondary study that provides an overview of the state of the art w.r.t. to support tools available for detecting/migrating third-party libraries in Android apps. To the best of our knowledge, none of the previous secondary studies has reviewed the literature from the point of view of support tools developed to aid green Android development. Most of the secondary studies discussed above have covered published work until 2015 or 2017 in the object-oriented paradigm, and many of the reviewed tools in these studies are now outdated/obsolete. Therefore, in this study we provide a different view of the literature by analyzing recently developed support tools for energy profiling, code optimization, refactoring, and third-party library detection or migration in Android development to improve energy efficiency in apps. We explore whether these support tools aid green Android development. We also provide an overview of the techniques used in these support tools.

7.3 Methodology

We conducted a systematic mapping study following the method described in [68]. First, we formulated research questions, and then based on those research questions we formulated two general search queries and conducted the search in the following online repositories for primary publications: IEEE Xplore, ACM digital library, Science Direct, and Springer. In this study, we cover publications from 2014 to June 2020, as from 2014 onwards the focus of many publications has been Android and energy-efficient app development, indicating a shift in research focus.

7.3.1 Research Questions

As the objective of this study is to analyze the current support tools available to improve custom code through detection/refactoring of code smell/energy bugs and to improve reusable code through detection/migration of third-party libraries in Android applications, we formulated the following research questions.

RQ1: *What state-of-the-art support tools have been developed to aid software practitioners in detecting/refactoring code smells/energy bugs in Android apps?*
RQ2: *What state-of-the-art support tools have been developed to aid software practitioners in detecting/migrating third-party libraries in Android apps?*
RQ3: *How do existing support tools compare to one another in terms of techniques they use for offering the support?*
RQ4: *How do existing support tools compare to one another in terms of the support they offer to practitioners for improving energy efficiency in Android apps?*

RQ1and RQ2 aim to classify publications based on the tools they offer. RQ3 aims to classify and analyze publications based on techniques used in the tool to offer support to developers. RQ4 deals with the characterization of all the identified tools in terms of the support (such as output or interface or availability) they offer to developers to aid green Android development.

7.3.2 Search Query

We derived search terms to use in our search query from the research questions of this study. We looked for alternatives to the search terms in publications we already knew and refined our search terms to return the most relevant publications. We used the "*" operator to cover possible variations on the selected search terms in the search query. The keyword "OR" was used to improve search coverage.

Based on our previous work [53], we improved our search query and extended our search in terms of publication years to include one more year. The first search query is designed to retrieve publications that provide a support tool to detect/refactor code smells/energy bugs in Android apps. The second search query is designed to retrieve publications that provide a support tool to detect/migrate third-party libraries in Android apps.

We did not use the search terms "mobile development," "apps," "optimization," "green," "sustainability," and "recommendation" in isolation as they were too high level and produced quite a large corpus consisting of a high number of irrelevant publications, while the search terms "resource leaks," "API," "tool," "framework," and "technique" were eliminated to avoid being too specific. The search queries were applied to popular online repositories (IEEE Xplore, ACM digital library, Science Direct, and Springer) to find a dataset of relevant primary publications. In each repository, based on available advanced search options, filters were applied to refine

Table 7.1 Search query filter

Filter	Value
Publication year	2014–2020 (up until June)
Content-type	Journal Article, Conference Paper

the query results. Applied filters are shown in Table 7.1. The search queries were applied to the titles, abstracts, and keywords of the publications.

Search Query 1
Android AND (energy OR code smell OR bug OR refactor OR correct* OR detect* OR optimiz* OR efficien*) AND NOT (environ* OR iot OR edu* OR hardware OR home)*

Search Query 2
(Android) AND ("third-party libr" OR "third-party Android lib*" OR "libr*") AND NOT (environ* OR iot OR indus* OR edu* OR hardware OR home)*

7.3.3 Screening of Publications

We first removed duplicate results and then defined inclusion, exclusion, and quality criteria for further screening of search results.

7.3.3.1 Duplicate Removal

The search results from online repositories were first loaded in Zotero[2] (an open-source reference management system) to create a dataset of relevant publications. Using the feature in Zotero, duplicate publications were removed from the dataset. Next, we manually applied inclusion, exclusion, and quality criteria to the remaining publications.

7.3.3.2 Inclusion Criteria

For inclusion, the selected publication should be a primary study generally related to the software engineering domain with a focus on third-party libraries or code smells or energy bugs in Android apps. A tool/automated technique for third-party library/code smell/energy bug detection, modification, or replacement was presented in the publication to support Android development. We considered only conference and journal articles published in English.

[2] https://www.zotero.org/

Table 7.2 Quality assessment criteria

ID	Description	Rating
1	Does the publication clearly state contributions that are directly related to third-party libraries/code smells/energy bugs in Android apps?	0.5
2	Is the contributions related to green in Android development?	0.5
3	Is the contributions a tool/automated technique that could be used in Android development/maintenance?	1
4	Is the research method adequately explained?	0.5
5	Are threats to validity and future research directions discussed separately?	0.5
	Total	3

7.3.3.3 Exclusion Criteria

Publications that were unrelated to Android development or third-party library/code smell/energy bugs in Android apps were excluded. The publications that focused only on hardware, environmental, security, privacy, networks, malware, clones, repackaging of apps, obfuscation issues, iOS, or present secondary data were also excluded. Work presented in a thesis or a book chapter is usually published in relevant journals or conferences as well. Therefore, doctoral symposium papers, magazine articles, book chapters, work-in-progress papers, and papers that were not in English were excluded as well.

7.3.3.4 Quality Criteria

The quality criteria applied to selected publications are shown in Table 7.2. Abstracts of the publications and structure of the publication were inspected for further quality assessment. If a quality rule was true for a publication, it was awarded full points; otherwise, no points were awarded. In case a rule partially applied to a publication, half points were awarded. After applying all five quality rules, the points were added to get a final quality score for a publication. A maximum quality score of 3 could be assigned to a publication. If a publication was below a total quality score of 2, it was removed from the results.

7.3.4 Classification and Analysis

To answer RQ1–RQ3, we identified the main keywords of the selected publications along with the commonly used terms in the abstracts to define categories of support tools. Research methodology and results of selected publications were additionally studied when needed. We kept extracted data in Excel spreadsheets for further processing. During data extraction, if there was a conflict of opinion, it was discussed among the authors until a consensus was reached. To answer RQ1 and

Table 7.3 Categories of support tools (RQ1)

ID	Category	Description
CP	Profiler	A software program that measures the energy consumption of an Android app or parts of apps
CD	Detector	A software program that only identifies and detects energy bugs/code smells in an Android app
CO	Optimizer	A software program that identifies energy bugs/code smells as well as refactor source code of an Android app to improve energy consumption

Table 7.4 Categories of support tools (RQ2)

ID	Category	Description
CI	Identifier	A software program that only identifies and detects third-party libraries in an Android app
CM	Migrator	A software program that identifies third-party libraries as well as helping in updating or migrating the third-party libraries (to an alternative library or version) in the source code of an Android app
CC	Controller	A software program that identifies third-party libraries to control, isolate, or de-escalate privileges and permissions granted to third-party libraries in an Android app

RQ2, a bottom-up merging technique was adopted to build our own classification schemes (see Tables 7.3 and 7.4). Once classification schemes were established, we extracted data from each selected publication to identify its main contribution and assigned the tool mentioned in the publication to a category based on the classification scheme. To answer RQ3, a classification scheme was needed to classify techniques used in support tools for offering support to aid Android development. We used the bottom-up approach to build this classification scheme by combining the specialized analysis methods/techniques into more generic higher-level techniques. The identified generic techniques along with their definitions are described in Tables 7.5 and 7.6. Once we had established the classification schemes, we extracted data from the abstract and research methodology of each selected publication and assigned it to a category defined in the classification schemes.

To answer RQ4, we extracted data form each selected publication to gather information about the kind of support the identified tool offers. We compare these tools based on the inputs of the tool, outputs of the tool, code smells/energy bugs/third-party libraries coverage, interface type, integrated development environment (IDE) support, and availability. In general, a code smell is defined as "a surface indication that usually corresponds to a deeper problem in the system" [69] and an energy bug is defined as an "error in the system (application, OS, hardware, firmware, external conditions or combination) that causes an unexpected amount of high energy consumption by the system as a whole" [70]. A third-party library is a reusable component related to specific functionality that can be integrated into the application to speed up the development process. A third-party library could be for advertising, analytics, Image, Network, Social Media, Utility, etc. [71]. In the light

Table 7.5 Categories of techniques used in support tools for code smell/energy bugs (RQ3)

ID	Technique	Definition
T1	Byte Code Manipulation	A technique that injects code in the Smali files of the app under test. The injected code is either a log statement or an energy evaluation function. These statements help determine the part of the source code that consumes a specific amount of energy at runtime.
T2	Code Instrumentation	A technique that instruments the app, using instrumented test cases that are capable of running specific parts of the app, in such a way that it is run in a specific environment while calling known methods/classes of the app under test. It uses finite state machines and device-specific power consumption details to measure energy.
T3	Logcat Analysis	A technique that uses system-level log files to obtain energy consumption information provided by the OS for the app under test. These logs are compared with application-level logs to give graphical information about the energy consumption of the app.
T4	Static Source Code Analysis	A technique that uses the source code of the app and analyzes it using one or a combination of the following methods: control flow graphs analysis, point-to-analysis, inter-procedural, intra-procedural, component call analysis, abstract syntax tree traversal, or taint analysis.
T5	Search-Based Algorithms	A technique that uses a multi-objective search algorithm to find multiple refactoring solutions and the most optimal solution is selected as final refactoring output by iteratively comparing the quality of design and energy usage.
T6	Dynamic Analysis	A technique based on the identification of information flow between objects at runtime for the detection of vulnerabilities in the app under test. It monitors the spread of sensory data during different app states.

of these definitions, we looked for Android-specific code smells, energy bugs, and third-party libraries in the studies.

7.4 Results

In this section, we present the result of the mapping study. The list of selected publications and additional details about code smells/energy bugs covered by support tools are shown in a separate file (additional materials).[3]

[3] Additional material: https://figshare.com/s/da429977adc4e928fd64

Table 7.6 Categories of techniques used in support tools for third-party libraries (RQ3)

ID	Technique	Description
T7	Feature Similarity	A technique that uses machine learning to extract code clusters or train classifiers by using feature hashing or similarity metrics or pattern digest or similarity digest on apps and third-party libraries code in order to identify and classify third-party libraries.
T8	Whitelist Comparison	A technique that compares third-party library names/versions/package information to whitelist in order to detect third-party libraries.
T9	API Hooking	A technique that intercepts or redirects API calls at various levels in order to regulate permission or policy-related operations.
T10	Module Decoupling	A technique to divide code into modules and extract code features such as package name, package structure, and inheritance relationships for clustering/classification to detect library.
T11	Process Isolation	A technique to isolate untrusted components in the operating system. This technique requires system-level modification.
T12	Class Profile Similarity	A technique to extract (strict or relaxed) profiles from libraries and apps code based on structural hierarchies. Based on similarity (exact or fuzzy) between these profiles library is detected.
T13	Collaborative Filtering	A technique to predict or recommend third-party libraries based on feature vectors and their similarity against a set of similar apps or neighborhood apps. It includes model-based approaches (such as matrix factorization), memory, and item-based approaches.
T14	Natural Language Processing	A technique used to identify or recommend third-party libraries based on textual descriptions. It includes techniques such as word embedding, skip-gram model, continuous bag-of-words model, domain-specific relational and categorical tag embedding, and topic modeling.

7.4.1 Results of Screening

Search Query 1 (Support Tools for Code Smell/Energy Bugs)
As a result of running search query 1 and applying filters (see Table 7.1) to search results, 2334 publications were found from the selected online repositories. These publications were loaded into the Zotero software for the screening and removal of duplicates, and the total number of publications was reduced to 2241 after duplicate removal. Inclusion and exclusion criteria were applied to the remaining publications, and the number was reduced to 575. We read abstracts of these publications and looked at the structure to assign them a quality score based on quality criteria. After applying the quality criteria, the number of selected publications was reduced to 24 (see Tables 7.7, 7.8, and 7.9).

Search Query 2 (Support Tools for Third-Party Libraries)
As a result of running search query 2 and applying filters (see Table 7.1) to search results, 545 publications were found from the selected online repositories. These publications were loaded into the Zotero software for the screening and removal of duplicates, and the total number of publications was reduced to 521 after duplicate

Table 7.7 Number of studies extracted per online repository (search query 1)

Sr.	Repo.	# of papers	Conference papers	Journal articles
1	IEEE Xplore	1170	910	260
2	ACM Digital library	483	459	24
3	Springer	595	362	231
4	Science Direct	86	4	82

Table 7.8 Number of articles per screening step (search query 1)

Sr.	Step in the screening of publications	# of publications
1	Search string results after applying filters	2334
2	Remove duplicates	2241
3	Apply inclusion and exclusion criteria	575
4	Apply quality criteria	24

Table 7.9 Quality score assigned to each selected publication (search query 1)

Publication ID	Quality score
P2, P11, P12, P14, P16, P17, P19, P20	2
P5, P6, P7, P13, P21, P24	2.25
P1, P4, P8, P9, P10, P15, P22, P23	2.5
P3, P18	2.75

Table 7.10 Number of studies extracted per online repository (search query 2)

Sr.	Repo.	# of papers	Conference papers	Journal articles
1	IEEE Xplore	312	296	12
2	ACM Digital library	177	157	20
3	Springer	28	22	6
4	Science Direct	28	0	28

Table 7.11 Number of articles after applying filters and screening steps (search query 2)

Sr.	Step in the screening of publications	# of publications
1	Search string results after applying filters	545
2	Remove duplicates	521
3	Apply inclusion and exclusion criteria	131
4	Apply quality criteria	27

removal. Inclusion and exclusion criteria were applied to the remaining publications and the number was reduced to 131. We read abstracts of these publications and looked at the structure to assign them a quality score based on quality criteria. After applying the quality criteria, the number of selected publications was reduced to 27 (see Tables 7.10, 7.11, and 7.12).

Table 7.12 Quality score assigned to each selected publication (search query 2)

Publication ID	Quality score
P31, P43	2
P26, P29, P33, P34, P35, P36, P39, P40, P48, P49	2.25
P25, P27, P28, P30, P32, P37, P38, P41, P42, P44, P45, P46, P47, P50, P51	2.5

Table 7.13 Distribution of studies in each category (search query 1)

ID	Selected publications	# Tools
CP	P6, P14, P16, P12, P13, P20, P19	7
CD	P1, P3, P4, P5, P8, P9, P7, P17	8
CO	P10, P11, P15, P18, P2, P21, P22, P23, P24	9

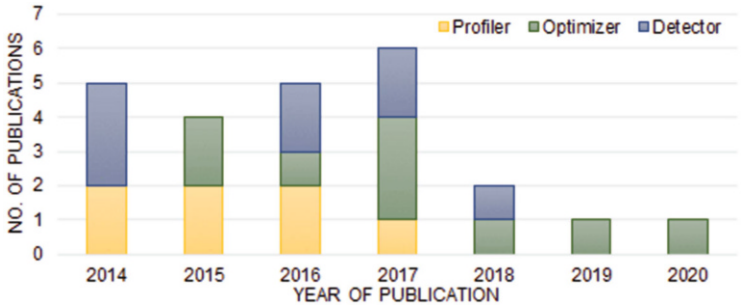

Fig. 7.1 Publications per year per category (search query 1)

7.4.2 Classification and Analysis

RQ1: What State-of-the-Art Support Tools Have Been Developed to Aid Software Practitioners in Detecting/Refactoring Code Smells/Energy Bugs in Android Apps?

To answer RQ1, the classification scheme defined in Table 7.3 (cf. Sect. 7.3.4) was used and the selected publications were divided into three categories, i.e., (1) "Profiler," (2) "Detector," and (3) "Optimizer," based on the support tool they offer to aid green Android development. Table 7.13 gives an overview of the distribution of selected publications in each category, along with the total number of tools in each category. Figure 7.1 shows the number of publications each year. The colors in the bars indicate the number of tools in each category each year from 2014 to 2020. We can see a decrease in the number of "Profiler" tools while there is an increase in the number of "Optimizer" tools. In 2019 and 2020 (until June), no new "Detector" tool was published.

Table 7.14 Distribution of publications in each category (search query 2)

ID	Selected publications	# Tools
CI	P26, P27, P29, P30, P31, P32, P33, P37, P40, P41, P42, P44, P47, P48, P49	16
CM	P35, P45, P50, P51	4
CC	P25, P28, P34, P36, P38, P39, P43, P46	7

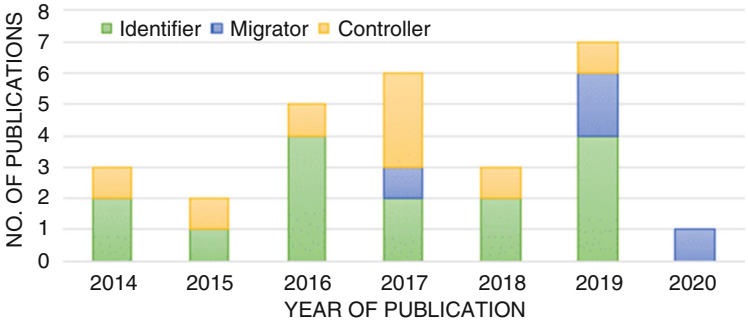

Fig. 7.2 Publications per year per category (search query 2)

Table 7.15 Overview of support tools (for code smell/energy bug detection and refactoring) showing the technique used for offering support to developers

Techniques						
Ct.	T1	T2	T3	T4	T5	T6
CP	P6, P16	P12, ‘P13, P19	P14, P20	–	–	P17
CD	–		–	P1, P3, P7, P8, P9, P4, P5	–	–
CO	–	–	–	P11, P21, P2, P10, P22, P23, P24	P15	P18

RQ2: What State-of-the-Art Support Tools Have Been Developed to Aid Software Practitioners in Detecting/Migrating Third-Party Libraries in Android Apps?

To answer RQ2, the classification scheme defined in Table 7.4 (cf. Sect. 7.3.4) was used and the selected publications were divided into the categories (1) “Identifier,” (2) “Migrator,” and (3) “Controller,” based on the support tool they offer to aid Android development. Table 7.14 gives an overview of the distribution of selected publications in each category, along with the total number of tools in each category. Figure 7.2 shows the number of publications each year. The colors in the bars indicate the number of tools in each category each year from 2014 to 2020 (until June). We can see at least one “Identifier” and “Controller” tool each year. In addition, we can see an increase in the number of “Migrator” tools in 2019 and 2020.

RQ3: How Do Existing Support Tools Compare to One Another in Terms of Techniques They Use for Offering the Support?

To answer RQ3, we identify techniques used in each tool for improving the energy efficiency of apps. Tables 7.15 and 7.16 give an overview of tools and techniques

Table 7.16 Overview of support tools (for third-party library detection and migration) showing the technique used for offering support to developers

Techniques								
Ct.	T7	T8	T9	T10	T11	T12	T13	T14
CI	P26, P42, P49, P33, P37, P40	P31, P47		P27, P29, P32, P44, P33, P37, P40, P47		P30, P41, P48		
CM				P45			P35, P51	P35, P50
CC			P34, P36, P38, P39, P43, P46	P28	P25			

along with reference to selected publications. Based on Table 7.15, we observed that no tool in any category used a combination of techniques. Each tool could be easily classified into exactly one category of techniques (defined in Sect. 7.3.4). However, in Table 7.16, many tools used a combination of techniques such as module decoupling and feature similarity, or collaborative filtering and natural language processing.

As a result of fine-tuning search query 1, we were able to identify three new "Optimizer" tools [P22, P23, and P24] which used static source code analysis to refactor and optimize the application code. See additional materials[3] for more details on techniques used in the "Profiler," "Detector," and "Optimizer" categories.

Identifier "Identifier" tools mostly used feature similarity or module decoupling or both techniques to detect third-party libraries. The authors of [P26] used similarity digests (which are similar to standard hashes) and compared them against a database consisting of original compiled code of third-party libraries. The authors of [P42] also used similarity digests to measure the similarity between data objects. The authors of [P49] used design pattern digests, fuzzy signatures, and fuzzy hash to match design patterns from app and library code. The authors of [P27 and P29] identified third-party libraries by decoupling an app into modules using package hierarchy clustering and clustering based on locality sensitive hashing, respectively. The authors of [P32 and P44] decoupled apps into modules to extract package dependencies for identifying third-party libraries. The authors of [P33, P37 and P40] used a combination of module decoupling and feature hashing/digests to provide a list of detected third-party libraries. The authors of [P47] used whitelist-based detection for non-obfuscated[4] apps and used motifs subgraph-based detection for obfuscated apps. The authors of [P31] used whitelist-based detection by comparing library name and package information against a list of commonly used third-party libraries. The authors of [P31, P41, and P48] extracted method signatures and

[4]Code obfuscation is used to conceal or obscure the code in order to avoid tempering.

package hierarchy structures from libraries to build profiles per library and used these profiles for third-party library identification.

Migrator "Migrator" tools mostly used a combination of collaborative filtering and natural language processing techniques. The authors of [P35] used collaborative filtering in combination with topic modeling (applied to the textual description in readme files). Based on results of topic modeling, similar apps were identified, and the set of third-party libraries extracted from these similar apps were then used to recommend libraries to developers. The authors of [P50] applied word embedding and domain-specific relational and categorical knowledge on stack overflow questions to recommend alternative libraries. The authors of [P51] used collaborative filtering and applied the matrix factorization approach to neutralize bias while recommending libraries. The authors of [P45] used the "LibScout" tool to extract library profiles. These profiles were then used to determine if a library version should be updated or not.

Controller "Controller" tools mostly used API hooking techniques to provide control over library privileges based on policy. The authors of [P34] intercepted and controlled framework APIs. The authors of [P36] intercepted system APIs to extract runtime library sequence information. The authors of [P38] tracked the execution entry of the module and all related asynchronous executions at thread level. The authors of [P39] used the tool "Soot Spark" to get call graphs in order to identify Android APIs that leak data (based on a given policy). The authors of [P43] used binder hooking, in-VM API hooking, and GOT (global offset table) hooking to regulate permission and file-related operation of third-party libraries. The authors of [P46] intercepted permissions protected calls and checked them against a compiled list of third-party libraries in order to regulate privileges. The authors of [P28] extracted code features and package information to train a classifier to detect libraries and grant them privileges. The authors of [P25] used system-level process isolation in order to separate third-party library privileges.

Techniques used to provide support by the various categories of support tools for detecting and refactoring code smells/energy bugs are as follows:

"Profiler" tools typically use a variety of techniques to measure energy consumption but none of the tools in this category uses static source code analysis.

Almost all "Detector" and "Optimizer" tools use static source code analysis of APK/SC based on a predefined set of rules.

Techniques used to provide support by the various categories of support tools for detecting and migrating third-party libraries are as follows:

"Identifier" tools use a variety of techniques for detecting third-party libraries. However, feature similarity and/or module decoupling techniques are more frequent.

Almost all "Migrator" tools used collaborative filtering and/or natural language processing techniques to recommend library migration.

Almost all "Controller" tools used API hooking techniques to control privileges/permissions related to third-party libraries.

RQ4: How Do Existing Support Tools Compare to One Another in Terms of the Support They Offer to Practitioners for Improving Energy Efficiency in Android Apps?

To answer RQ4, we first list all the support tools for code smell/energy bug detection/correction (see Table 7.17) and compare them in terms of input, output, user interface, integrated development environment (IDE) integration, availability, and code smell/energy bug coverage. Second, we list all the support tools for detecting/migrating third-party libraries (see Table 7.18) and compare them in terms of input, output, library coverage, user interface, availability, and IDE integration support.

In Tables 7.17 and 7.18, the "input" column provides information about the input for each tool. The "output" column provides information about the support the tool offers based on the input. The "UI" column provides information about the user interface of the tool. The "open source" column provides information about tool availability for usage/extension. The "IDE" column (in Table 7.17) provides information about the IDE integration capability of tools. The "TPL Type" column (in Table 7.18) provides information about the third-party library (TPL) coverage of the tool.

Support Tools for Code Smell/Energy Bug Detection and Refactoring

In Table 7.17, we provide a list of all the tools identified in the "Profiler," "Detector," and "Optimizer" categories. As a result of fine-tuning search query 1, we were able to identify three new "Optimizer" tools [P22, P23, P24] that were not included in our previous work [53]. For all the 24 tools listed in Table 7.17 we provide additional information related to interface, availability, and IDE integration that was not included in previous work [53].

Studies in the category "Profiler" offer support to the practitioners by providing tools that can measure the energy consumed by the whole/parts of an app or device sensors used in the apps. The measured information is usually presented to practitioners as graphs for energy consumption over time. Studies in the "Profiler" category do not recommend when, where, and how practitioners can use the information from these graphs during development to improve the energy consumption of their apps. Studies in the category "Detector" offer support to practitioners by developing tools that present as output lists of energy bugs/code smells causing a change in energy consumption of apps. Studies in the category "Optimizer" offer support to practitioners by developing tools that present as output refactored source code of apps optimized for energy. The studies in this category do not explicitly give the recommendation to the developers about how to optimize the source code for energy efficiency as the tools automatically refactor the code.

Out of the 24 tools listed in Table 7.17, only seven are open source. Out of the seven open-source tools, three are "Detector" tools, and four are "Optimizer" tools. Most of the tools do not offer IDE integration. Four tools in "Optimizer" category support integration with Eclipse IDE [P11, P18, P21, P24] while one tool [P22] supports integration with Android Studio IDE. Out of 24 tools, 12 offer command-line interface (CMD) [P1, P3, P5, P9, P7, P10, P13, P15, P17, P20, P22, P23], eight

Table 7.17 List of support tools in "Profiler," "Detector," and "Optimizer" categories along with information about their inputs and outputs, user interface, IDE support, and availability

Ct.	Tool	Input	Output	UI	IDE	Open source	ID
CP	Orka	APK	ECG	GUI	No	No	P6
	SEPIA	AE	ECG	GUI	No	No	P12
	Mantis	PBC	Program CRC predictors	CMD	No	No	P13
	AEP*	SL, PID via ADB	ECG	GUI	No	No	P14
	E-Spector	SL, AL via ADB	ECG	GUI	No	No	P16
	SEMA	PID, MVC	Log of EC	CMD	No	No	P20
	Keong et. al	SC	ECG	GUI + CMD	No	No	P19
CD	Wu et al.	SC	List of energy bugs	CMD	No	No	P1
	Kim et al.	PBC	List of energy bugs	CMD	No	No	P3
	Statedroid	APK	List of energy bugs	CMD	No	No	P5
	PatBugs	SC	List of detected warnings	NS	No	No	P8
	SAAD	APK	List of energy bugs	CMD	No	No	P9
	aDoctor	SC	List of code smells	GUI + CMD	No	Yes	P4
	GreenDroid	PBC, CF	List of energy bugs + severity level	CMD	No	Yes	P17
	Paprika	APK, PM	List of code smells	CMD	No	Yes	P7
CO	DelayDroid	APK	Refactored APK	NS	No	No	P2
	HOT-PEPPER	APK	Most energy efficient APK, Refactored SC, and List of refactoring	CMD	No	Yes	P10
	Asyncdroid	SC	Refactored SC	GUI	Eclipse	No	P11
	EARMO	APK	Refactored APK	CMD	No	Yes	P15
	EnergyPatch	APK	Refactored APK	GUI	Eclipse	No	P18
	Nguyen et al.	SC	Refactored SC	GUI	Eclipse	No	P21
	Chimera	SC	Refactored APK	CMD	Android Studio	No	P22
	ServDroid	APK	Refactored APK	CMD	No	Yes	P23
	Leafactor	SC	Refactored APK file	GUI	Eclipse	Yes	P24

Ct. category, *SC* source code, *APK* android package kit, *PBC* program byte code, *SL* system log files, *AL* application log files, *PID* process ID, *ADB* android debug bridge, *CRC* computational resource consumption, *AE* application events, *CF* configuration files, *MVC* measurements of voltage and current, *ECG* energy consumption graph, *SM* software metrics values, *PM* playstore metadata, *GUI* graphical user interface, *CMD* command line, *EC* energy consumption

Table 7.18 List of support tools in "Identifier," "Migrator," and "Controller" categories along with information about their inputs and outputs, library coverage, UI, and availability

Ct.	Tool	Input	Output	TPL Type	UI	Open source	Ref
CI	Duet	APK	Library integrity pass/fail ratio	Java	NS	No	P26
	AdDetect	APK	List of detected TPLs	Java-Ad	NS	No	P27
	AnDarwin	APK	Detect and exclude TPLs + Set clone or rebranded apps	Java	NS	No	P29
	LibScout	TPL .jar/. aar + APK	Presence of given TPL based on similarity score	Java	CMD	Yes	P30
	DeGuard	APK	De-obfuscated APK (containing detected TPLs)	Java	GUI[a]	Yes	P31
	LibSift	APK	List of detected TPLs	Java	NS	No	P32
	LibRadar	APK	List of detected TPLs sorted by popularity + info about TPLs	Java	GUI[a]	Yes	P33
	LibD	APK	List of detected TPLs	Java	CMD	Yes	P37
	Ordol	APK	List of detected TPL versions + similarity score.	Java	NS	No	P40
	LibPecker	TPL name + APK	Presence of given TPL based on the similarity score	Java	NS	No	P41
	Orlis	APK	List of detected TPLs	Java	NS	Yes	P42
	PanGuard	APK	List of detected TPLs	Java	GUI[a]	No	P44
	He et al.	APK	List of detected TPLs + risk assessment	Java	NS	No	P47
	Feichtner et al.	APK/TPL	List of detected TPLs and versions + similarity score	Java	CMD[b]	Yes	P48
	DPAK	APK/ Android jar	List of detected TPLs	Java	CMD[b]	No	P49
CM	AppLibRec	SC	List of recommended TPLs	Java	NS	No	P35
	Appcommune	APK	Tailored app without TPLs and updated/ customized TPLs	Java	GUI[c]	No	P45
	SimilarTech	TPL name	List of recommended TPLs + information about usage	Java	GUI[a]	No	P50
	LibSeek	APK	List of recommended TPLs	Java	NS	No	P51

(continued)

Table 7.18 (continued)

Ct.	Tool	Input	Output	TPL Type	UI	Open source	Ref
CC	NativeGuard	APK	Split original APK into Service APK and Client APK	Native	CMD	No	P25
	Pedal	APK	Repackaged APK with privilege de-escalated for detected TPLs	Java-Ad	GUI[c]	No	P28
	LibCage	SC + list of permissions required by TPLs	Deny unnecessary TPL permission on runtime	Java+ Native	NS	No	P34
	Zhan et al.	SC + Policy	Grant or deny permissions to TPLs based on policy	Java	NS	No	P36
	Perman	APK	Grant or deny permissions to TPLs based on policy	Java	GUI[c]	No	P38
	SurgeScan	TPL bytecode + Android.jar + policy	Dex and jar files of TPL with the policy implemented	Java	NS	No	P39
	AdCapsule	SC + policy	Grant or deny permissions to TPLs based on policy	Java-Ad	NS	No	P43
	Reaper	APK	Grant or deny permissions to TPLs based on user preference	Java + Native	GUI[c]	Yes	P46

Ct. category, *UI* user interface, *SC* source code, *APK* android package kit, *TPL* third-party libraries, *GUI* graphical user interface, *CMD* command-line interface, *NS* not specified in publication
[a]Web service
[b]Executable jar
[c]App on Android device

tools offer graphical user interface (GUI) [P6, P11, P12, P14, P16, P18, P21, P24], and two tools offer both [P4, P19], while for the rest of them information about interface is not specified in the publications.

See additional material[3] for details about definitions of code smells/energy bugs covered by tools in the "Detector" and "Optimizer" categories. Figure 7.3 shows the Android energy bug coverage of tools in the "Detector" and "Optimizer" categories. The Android energy bugs are shown on the horizontal axis. The percentage of tools in the "Detector" and "Optimizer" categories covering Android energy bugs is shown on the vertical axis. We can see that Android energy bugs "TMV," "TDL," "UL," "UP," and "VBS" are detected by 13% of the tools, whereas "RL," "WB," and "NCD" are detected by 75%, 50%, and 38% of the tools in the "Detector" category respectively. None of the tools in the "Optimizer" category covers. "TMV," "TDL,"

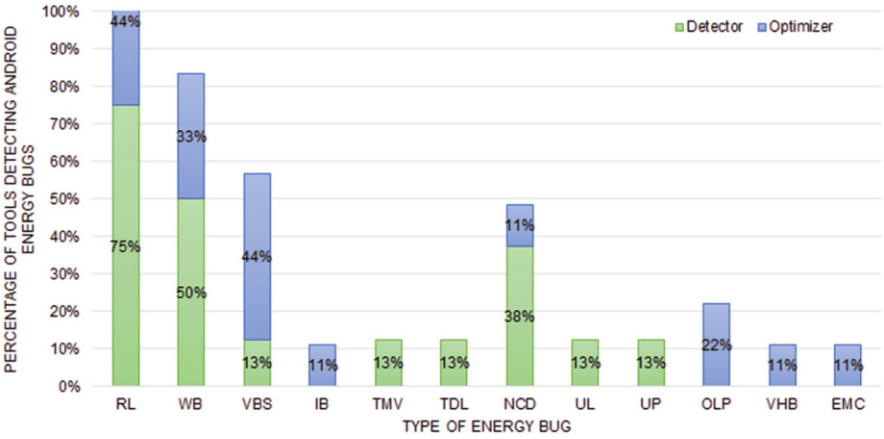

Fig. 7.3 Percentage of the tools in "Detector" and "Optimizer" categories that can detect Android energy bugs (*RL* resource leak, *WB* wake-lock bug, *VBS* vacuous background services, *IB* immortality bug, *TMV* too many views, *TDL* too deep layout, *NCD* not using compound drawables, *UL* useless leaf, *UP* useless parent, *OLP* obsolete layout parameter, *VHB* view holder bug, *EMC* excessive method calls)

"UL," and "UP" energy bugs. On the other hand, energy bugs "IB," "OLP," "VHB," and "EMC" are covered by tools in the "Optimizer" category, whereas none of the tools in the "Detector" category covers them. "RL" and "VBS" energy bugs are detected by 44% of the tools in the "Optimizer" category.

Figure 7.4 shows the Android code smell coverage of tools in the "Detector" and "Optimizer" categories. The Android code smells are shown on the vertical axis. The percentage of tools in the "Detector" and "Optimizer" categories covering Android code smells is shown on the horizontal axis. We can see that Android code smells "ERB" and "VHP" are not detected by any tool in the "Detector" category, whereas "LWS," "LC," "RAM," "PD," "ISQLQ," "IDFP," "DW," "DR," and "DTWC" are not detected by any of the tools in the "Optimizer" category. Android code smells such as "IOD," "HBR," "HSS," "HAT," "IWR," "UIO," "BFU," "UHA," "LWS," "LC," "SL," "RAM," "PD," "NLMR," "MIM," "LT," "IDS," "IDFP," "DW," "DR," and "DTWC" are detected by 13–25% of the tools in the "Detectors" category.

Typical support given by the various categories of support tools for detecting and refactoring code smells/energy bugs are as follows:

"Profiler" tools support developers by visualizing the energy consumption of the whole app or parts of it.

"Detector" tools support developers with lists of energy bugs and code smells to be manually fixed by the developer for energy improvement.

"Optimizer" tools support developers by automatically refactoring APK/SC versions based on predefined rules.

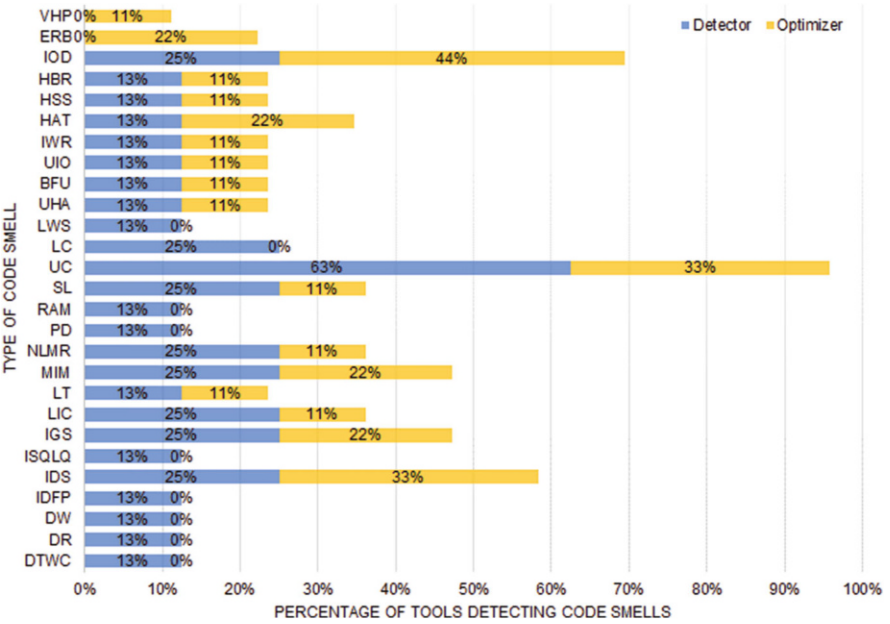

Fig. 7.4 Percentage of code smells detected by each tool in "Detector" and "Optimizer" categories. *DTWC* data transmission without compression, *DR* debuggable release, *DW* durable wake-lock, *IDFP* inefficient data format and parser, *IDS* inefficient data structure, *ISQLQ* inefficient SQL query, *IGS* internal getter and setter, *LIC* leaking inner class, *LT* leaking thread, *MIM* member ignoring method, *NLMR* no low memory resolver, *PD* public data, *RAM* rigid alarm manager, *SL* slow loop, *UC* unclosed closeable, *LC* lifetime containment, *LWS* long wait state, *UHA* unsupported hardware acceleration, *BFU* bitmap format usage, *UIO* UI overdraw, *IWR* invalidate without rect, *HAT* heavy AsyncTask, *HSS* heavy service start, *HBR* heavy broadcast receiver, *IOD* init ONDraw, *ERB* early resource binding, *VHP* view holder pattern

Support Tools for Third-Party Library Detection and Migration

In Table 7.18, we provide a list of all the tools identified in the "Identifier," "Migrator," and "Controller" categories. Publications in the category "Identifier" offer support to practitioners by providing tools that detect third-party libraries present in the apps. The information is usually presented to practitioners as a list of detected libraries along with their version/similarity scores. Publications in the category "Migrator" offer support to practitioners by developing tools that present as output lists of recommended third-party libraries. Publications in the category "Controller" offer support to practitioners by developing tools that present as output policy-based privilege/permission control over third-party libraries. Most tools in the "Identifier," "Migrator," and "Controller" categories provide coverage for all types (advertisement, social, network, billing, analytics, etc.) of Java-based third-party libraries. Some tools such as AdDetect (CI) or Pedal (CC) cover only the advertisement-related third-party libraries. NativeGuard (CC) provides coverage for only native third-party libraries. Reaper (CC) and LibCage (CC) provide coverage for native and Java-based third-party libraries. For many tools listed in

Table 7.18, interface type was not specified in publications, while others provide either a command-line interface (CMD) or a graphical user interface (GUI). Out of the 27 tools listed in Table 7.18, only seven tools are open source. Out of the seven open-source tools, six tools [P30, P31, P33, P37, P42, P48] are "Identifier" tools and one tool [P46] is a "Controller" tool. None of the tools listed in Table 7.18 provides IDE integration support.

> The tools in the "Identifier," "Migrator," and "Controller" categories do not detect/update/control/migrate third-party libraries to optimize the source code of Android applications for energy efficiency.

Typical support given by the various categories of support tools for detecting and migrating third-party libraries are as follows:

> "Identifier" tools support developers by detecting third-party libraries present in apps.
>
> "Migrator" tools support developers with lists of recommended third-party libraries along with the mapping information of these libraries for updating/migrating them.
>
> "Controller" tools support developers by separating third-party library privileges from the app privileges based on policy defined by developers.

7.5 Discussion

In this section, we discuss the results of the mapping study to identify future research opportunities.

7.5.1 Support Tools for Code Smell/Energy Bug Detection and Refactoring

We observed that most of the support tools in the "Profiler," "Detector," and "Optimizer" categories are not open source, making them inaccessible to many developers. On top of that, most of these support tools do not support IDE integration. Due to the rapid development process of Android applications, developers are more likely to use tools that are integrated with the IDEs and share the same interface design. The current state-of-the-art tools could be extended to integrate with other industrially famous code analyzers like Android Lint, Check Style, Find Bugs and PMD. Each tool in "Detector" and "Optimizer" category provided a limited coverage over Android-specific code smells/energy bugs. The industry relevance of the current state-of-the-art support tools might not be obvious because they are not evaluated in industrial settings. In principle, if developers spent time and effort to learn one such tool they still might not be able to identify many code smells/energy bugs in their code, unless they use a combination of these tools to get complete coverage. Most tools in the "Detector" and "Optimizer" categories used static source code analysis, which indicates that dynamic issues such as those related to

asynchronous tasks are not covered by these tools. For the development of better support tools, hybrid techniques encompassing both dynamic and static analysis could be used. In addition, non-intrusive techniques could be used to collect software metrics for identifying code smells/energy bugs. The results from the selected publications could be expanded to include cross-project predictions and corrections for energy bugs. Analysis and inclusion of multi-threaded programming approaches in the experiments could be another direction for future researchers.

7.5.2 Support Tools for Third-Party Library Detection and Migration

We observed that none of the support tools in the "Identifier," "Migrator," and "Controller" categories provides support for IDE integration and many of these tools are also not open source, making them inaccessible to developers. We also observed the none of the support tools in these categories offers any support to developers to aid green Android development. One possible reason could be that so far research related to third-party library identification is mostly used in clone detection, detection of rebranded/similar/malicious apps, and detection of issues related to security, privacy, or data leaks (see related work). However, there is a gap in the literature regarding support tools that identify/update/recommend third-party libraries to aid green Android development. Anwar et al. [52] have investigated the energy consumption of third-party libraries in Android applications, indicating that the energy consumption of alternative third-party libraries varies significantly in various use cases. Rasmussen et al. [66] showed that blocking advertisements in Android apps reduces energy consumption. However, these studies have only focused on a small subset of network- and advertisement-related libraries. Energy consumption of other types of libraries such as social, analytical, or utility has not yet been explored, and merits further research. Data from such studies could be used by tool developers to recommend energy-efficient libraries to developers during development. Support tools in the "Migrator" category are good candidates for this type of research as the collaborative filtering and natural language processing techniques could supplement the data gathered from energy reading of third-party libraries. Such information could be useful in mapping the function of one library to another alternative library for a smooth migration. Support tools in the "Identifier" category generally use two techniques: (a) whitelist-based and (b) similarity-based. Tools that used whitelist-based approaches are fast due to a smaller feature set, and thus could perform better in large-scale analysis. However, this technique cannot identify third-party libraries without prior knowledge. On the other hand, tools that use similarity-based approaches such as feature hashing use a larger feature set and can identify third-party libraries without prior knowledge. Due to the extended feature set, these tools might be more accurate but time-consuming. Many tools in the "Identifier" category (such as "LibD," "LibScout," "LibRadar," or "AdDetect") consider code

obfuscation during library detections in order to give accurate results. However, not many tools are resilient against code shrinking as they rely on package hierarchies. Support tools in the "Controller" category rely on API hooking techniques which separate libraries from app code. Such tools could also benefit from using an access control list to split privileges. Because current techniques require system-level changes, this makes the deployment of "Controller" tools difficult.

7.6 Threats to Validity

The search queries and classification of selected publications could be biased by the researcher's knowledge. We mitigated this threat by defining the inclusion, exclusion, and quality criteria for the selection of the publications. Conflicting opinions were discussed among authors of this study until a consensus was reached. In order to avoid false-positives and false-negatives in the search results, we used the wildcard character (*) to maximize coverage and the keyword "AND NOT" to remove irrelevant studies. We did not use the terms "energy" or "efficiency" in combination with "Android" in the second search query, as we had already executed this combination in search query 1. The results of the search strings were manually checked and further refined by the authors. Online repositories continuously update their databases to include new publications, and therefore executing the same queries might yield some additional results that were not included in this study. We already knew about many relevant studies and we recaptured almost 90% of them when we executed the search queries. On each online repository the search mechanism is slightly different and we tried to keep the queries as consistent as possible, but there might be a slight difference due to the difference in search mechanism provided by different online repositories. Some selected publications use the terms code smells and energy bugs interchangeably, which could affect the classification. To mitigate this threat, we used the selected definitions (cf. Sect. 7.3.4) for code smells and energy bugs to correctly classify the studies in the right category.

We have excluded publications that did not focus on Android development yet still contributed a tool for detecting/recommending third-party libraries. Maven central repository contains a huge quantity of Java-based third-party libraries that can be used in any Java-based application. However, in this study, we focused particularly on the support tools for energy profiling, code optimization and refactoring of code smells/energy bugs, and detection/migration of third-party libraries to help aid green Android development. Other types of support tools, such as tools for style checking, interface optimization, test generation, requirement engineering, and code obfuscation, were not in the scope of this study. Therefore, while applying inclusion/exclusion criteria, we filtered support tools such as "LibFinder," LibCPU, CrossRec, and RAPIM [72–75]. These tools could identify/recommend third-party libraries but they were not designed to be used specifically with Android applications. We plan to cover such tools in future work.

7.7 Conclusions

We conducted a mapping study to give an overview of the state of the art and to find research opportunities with respect to support tools available for green Android development. Based on our analysis we identified tools for detecting/refactoring code smells/energy bugs, which were classified into three categories: (1) "Profiler," (2) "Detector," and (3) "Optimizer." Additionally, we identified tools for detecting/migrating third-party libraries in Android applications, which were classified into (1) Identifier, (2) Migrator, and (3) Controller categories. The main findings of this study are that most "Profiler" tools provide a graphical representation of energy consumption over time. Most "Detector" tools provide a list of energy bugs/code smells to be manually corrected by a developer for the improvement of energy. Most "Optimizer" tools automatically convert original APK/SC into a refactored version of APK/SC. Tools in the "Identifier," "Migrator," and "Controller" categories do not provide support to developers to optimize code w.r.t. energy consumption. The most typical technique in the "Detector" and "Optimizer" categories was static source code analysis using a predefined set of code smells and rules. The most typical techniques in the "Identifier" category were module decoupling and feature similarity, while in the "Migrator" and "Controller" categories, API hooking and collaborative filtering in combination with natural language processing were used, respectively.

Acknowledgments This work was supported by the Estonian Center of Excellence in ICT research (EXCITE), the group grant PRG887 funded by the Estonian Research Council, and the Estonian state stipend for doctoral studies.

References

1. GeSI (2015) #SMARTer2030 ICT solutions for 21st century challenges. Accessed 06 Jun 2020. http://smarter2030.gesi.org/downloads/Full_report.pdf
2. Acar H (2017) Software development methodology in a Green IT environment. Université de Lyon
3. Calero C, Piattini M (2015) Introduction to green in software engineering. In: Calero C, Piattini M (eds) Green in software engineering. Springer International Publishing, Cham, pp 3–27
4. Chauhan NS, Saxena A (2013) A green software development life cycle for cloud computing. IT Prof 15(1):28–34. https://doi.org/10.1109/MITP.2013.6
5. Federal Ministry for Economic Affairs and Energy (2014) Energy-efficient ICT in practice: planning and implementation of GreenIT measures in data centres and the office
6. Jagroep E, van der Werf JM, Brinkkemper S, Blom L, van Vliet R (2017) Extending software architecture views with an energy consumption perspective. Computing 99(6):553–573. https://doi.org/10.1007/s00607-016-0502-0
7. Kumar S, Buyya R (2012) Green cloud computing and environmental sustainability. Harnessing Green It Princ Pract:315–339. https://doi.org/10.1002/9781118305393.ch16
8. Oyedeji S, Seffah A, Penzenstadler B (2018) A catalogue supporting software sustainability design. Sustainability 10(7):2296. https://doi.org/10.3390/su10072296

9. Gupta PK, Singh G (2012) Minimizing power consumption by personal computers: a technical survey. Int J Inf Technol Comput Sci 4(10):57–66. https://doi.org/10.5815/ijitcs.2012.10.07
10. Kern E et al (2018) Sustainable software products—towards assessment criteria for resource and energy efficiency. Futur Gener Comput Syst 86(3715):199–210. https://doi.org/10.1016/j.future.2018.02.044
11. Murugesan S, Gangadharan GR (2012) Green IT: an overview. In: Murugesan S, Gangadharan GR (eds) Harnessing green IT: principles and practices. Wiley, pp 1–21
12. Egham (2018) Gartner says worldwide end-user device spending set to increase 7 percent in 2018; global device shipments are forecast to return to growth. Gartner, Press Releases. Accessed 11 Feb 2019. https://www.gartner.com/en/newsroom/press-releases/2018-04-05-gartner-says-worldwide-end-user-device-spending-set-to-increase-7-percent-in-2018-global-device-shipments-are-forecast-to-return-to-growth
13. Penzenstadler B, Femmer H (2013) A generic model for sustainability with process- and product-specific instances. In: Proceedings of the 2013 Workshop on Green by Software Engineering, pp 3–7. doi:https://doi.org/10.1145/2451605.2451609
14. Raturi A, Tomlinson B, Richardson D (2015) Green software engineering environments. In: Green in software engineering. Springer International Publishing, pp 31–59
15. Banerjee A, Chong LK, 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 - FSE, vol 16–21-Nov, pp 588–598, doi: https://doi.org/10.1145/2635868.2635871
16. Allix K, Bissyandé TF, Klein J, Le Traon Y (2016) AndroZoo: collecting millions of Android apps for the research community. In: Proceedings of the 13th international workshop on mining software repositories - MSR, May 2016, pp 468–471, doi: https://doi.org/10.1145/2901739.2903508
17. Anwar H, Pfahl D (2017) Towards greener software engineering using software analytics: a systematic mapping. In: Proceedings of the 43rd Euromicro conference on software engineering and advanced applications -SEAA, Aug 2017, pp 157–166, doi: https://doi.org/10.1109/SEAA.2017.56
18. Martin W, Sarro F, Jia Y, Zhang Y, Harman M (2017) A survey of app store analysis for software engineering. IEEE Trans Softw Eng 43(9):817–847. https://doi.org/10.1109/TSE.2016.2630689
19. Oliveira W, Oliveira R, Castor F (2017) A study on the energy consumption of android app development approaches. In: Proceedings of the IEEE/ACM 14th international conference on mining software repositories - MSR, May 2017, pp 42–52, doi: https://doi.org/10.1109/MSR.2017.66
20. Rawassizadeh R (2010) Mobile application benchmarking based on the resource usage monitoring. Int J Mob Comput Multimed Commun 1(4):64–75. https://doi.org/10.4018/jmcmc.2009072805
21. Viennot N, Garcia E, Nieh J (2014) A measurement study of google play. ACM SIGMETRICS Perform Eval Rev 42(1):221–233. https://doi.org/10.1145/2637364.2592003
22. Wang H et al (2017) An explorative study of the mobile app ecosystem from app developers' perspective. In: Proceedings of the 26th international conference on World Wide Web, pp 163–172, doi:https://doi.org/10.1145/3038912.3052712
23. Wang H et al (2018) Beyond Google play: a large-scale comparative study of Chinese Android App Markets. ArXiv, vol 1810.07780, Sep 2018. http://arxiv.org/abs/1810.07780
24. Ardito L, Procaccianti G, Torchiano M, Migliore G (2013) Profiling power consumption on mobile devices. In: Proceedings of the third international conference on smart grids, green communications and IT Energy-aware Technologies, pp 101–106
25. Azevedo L, Dantas A, Camilo-Junior CG. DroidBugs: an android benchmark for automated program repair. ArXiv, vol abs/1809.0, 2018 [Online]. http://arxiv.org/abs/1809.07353

26. Chung YF, Lin CY, King CT (2011) ANEPROF: energy profiling for android java virtual machine and applications. In: Proceedings of the international conferences on parallel and distributed systems - ICPADS, pp 372–379, doi: https://doi.org/10.1109/ICPADS.2011.28
27. Kansal A, Zhao F (2008) Fine-grained energy profiling for power-aware application design. ACM SIGMETRICS Perform Eval Rev 36(2):26. https://doi.org/10.1145/1453175.1453180
28. Pathak A, Hu YC, Zhang M (2012) Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. EuroSys, pp 29–42, Accessed 04 Apr 2018. https://www.cse.iitb.ac.in/~mythili/teaching/cs653_spring2014/references/energy-eprof-tool.pdf
29. Banerjee A, Roychoudhury A (2016) Automated re-factoring of Android apps to enhance energy-efficiency. In: Proceedings of the international workshop on mobile software engineering and system - MOBILESoft, pp 139–150, doi: https://doi.org/10.1145/2897073.2897086
30. Fernandes TS, Cota E, Moreira AF (2014) Performance evaluation of android applications: a case study. In: Proceedings of the Brazilian symposium on computing system engineering, Nov 2014, vol 1998-Jan, pp 79–84, doi: https://doi.org/10.1109/SBESC.2014.17
31. Fowler M, Beck K (1999) Refactoring: improving the design of existing code. Addison-Wesley
32. Hecht G, Rouvoy R, Moha N, Duchien L (2015) Detecting antipatterns in android apps. In: Proceedings of the 2nd ACM international conference on mobile software engineering and systems, MOBILESoft, Sep 2015, pp 148–149, doi: https://doi.org/10.1109/MobileSoft.2015.38
33. Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A (2017) Lightweight detection of Android-specific code smells: the aDoctor project. In: Proceedings of the 24th IEEE international conference software analysis evolution and reengineering - SANER, pp 487–491. doi: https://doi.org/10.1109/SANER.2017.7884659
34. Rasool G, Ali A (2020) Recovering android bad smells from android applications. Arab J Sci Eng 45(4):3289–3315. https://doi.org/10.1007/s13369-020-04365-1
35. Xu B, An L, Thung F, Khomh F, Lo D (2020) Why reinventing the wheels? An empirical study on library reuse and re-implementation. Empir Softw Eng 25(1):755–789. https://doi.org/10.1007/s10664-019-09771-0
36. Wang H, Guo Y (2017) Understanding third-party libraries in mobile app analysis. In: Proceedings of the IEEE/ACM 39th international conference on software engineering companion, pp 515–516, doi: https://doi.org/10.1109/ICSE-C.2017.161
37. Zhan J, Zhou Q, Gu X, Wang Y, Niu Y (2017) Splitting third-party libraries' privileges from android apps. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol 10343 LNCS, Springer, pp 80–94
38. Gkortzis A, Feitosa D, Spinellis D (2019) A double-edged sword? Software reuse and potential security vulnerabilities. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol 11602 LNCS, pp 187–203, doi: https://doi.org/10.1007/978-3-030-22888-0_13
39. Ikram M, Vallina-Rodriguez N, Seneviratne S, Kaafar MA, Paxson V (2016) An analysis of the privacy and security risks of android VPN permission-enabled apps. In: Proceedings of the ACM SIGCOMM internet measurement conference - IMC, vol 14–16-Nov, pp 349–364, doi: https://doi.org/10.1145/2987443.2987471
40. Mazuera-Rozo A, Bautista-Mora J, Linares-Vásquez M, Rueda S, Bavota G (2019) The Android OS stack and its vulnerabilities: an empirical study. Empir Softw Eng 24(4):2056–2101. https://doi.org/10.1007/s10664-019-09689-7
41. Ogawa H, Takimoto E, Mouri K, Saito S (2018) User-side updating of third-party libraries for android applications. In: Proceedings of the sixth international symposium on computing and networking workshops - CANDARW, Nov 2018, pp 452–458, doi: https://doi.org/10.1109/CANDARW.2018.00088
42. Binns R, Zhao J, Van Kleek M, Shadbolt N (2018) Measuring third-party tracker power across web and mobile. ACM Trans Internet Technol 18(4). doi: https://doi.org/10.1145/3176246

43. Fu J, Zhou Y, Liu H, Kang Y, Wang X (2017) Perman: fine-grained permission management for android applications. In: Proceedings of the IEEE 28th international symposium on software reliability engineering - ISSRE, Oct 2017, vol 2017-Oct, pp 250–259, doi: https://doi.org/10.1109/ISSRE.2017.38
44. Gao X, Liu D, Wang H, Sun K (2016) PmDroid: permission supervision for android advertising. In: Proceedings of the IEEE symposium on reliable distributed systems, vol 2016-Jan, pp 120–129, doi: https://doi.org/10.1109/SRDS.2015.41
45. Jin H et al. (2018) Why are they collecting my data?. In: Proceedings of the ACM on interactive, mobile, wearable and ubiquitous Techniques, Dec 2018, vol 2(4), pp 1–27, doi:https://doi.org/10.1145/3287051
46. Wang H, Li Y, Guo Y, Agarwal Y, Hong JI (2017) Understanding the purpose of permission use in mobile apps. ACM Trans Inf Syst 35(4). https://doi.org/10.1145/3086677
47. Chen K, Liu P, Zhang Y (2014) Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: Proceedings of the international conference on software engineering, no 1, pp 175–186, doi: https://doi.org/10.1145/2568225.2568286
48. Li L, Bissyandé TF, Wang HY, Klein J (2019) On identifying and explaining similarities in android apps. J Comput Sci Technol 34(2):437–455. https://doi.org/10.1007/s11390-019-1918-8
49. Soh C, Tan HBK, Arnatovich YL, Wang L (2015) Detecting clones in android applications through analyzing user interfaces. In: Proceedings of the IEEE 23rd international conference on program comprehension, May 2015, pp 163–173, doi:https://doi.org/10.1109/ICPC.2015.25
50. Yuan L (2016) Detecting similar components between android applications with obfuscation. In: Proceedings of the 5th international conference on computer science and networking technologies - ICCSNT, Dec 2016, pp 186–190, doi:https://doi.org/10.1109/ICCSNT.2016.8070145
51. Zhang Y, Ren W, Zhu T, Ren Y (2019) SaaS: a situational awareness and analysis system for massive android malware detection. Futur Gener Comput Syst 95:548–559. https://doi.org/10.1016/j.future.2018.12.028
52. Anwar H, Demirer B, Pfahl D, Srirama SN (2020) Should energy consumption influence the choice of Android third-party HTTP libraries?. In: Proceedings of the IEEE/ACM 7th International conference on mobile software engineering and systems, MOBILESoft, pp 87–97. doi: https://doi.org/10.1145/3387905.3392095
53. Fatima I, Anwar H, Pfahl D, Qamar U (2020) Tool support for green android development: a systematic mapping study. In: Proceedings of the 15th international conference on software technologies - ICSOFT, pp 409–417
54. Fontana FA, Mariani E, Mornioli A, Sormani R, Tonello A (2011) An Experience report on using code smells detection tools. In: Proceedings of the IEEE fourth international conference on software testing, verification and validation workshops, Mar 2011, pp 450–457, doi:https://doi.org/10.1109/ICSTW.2011.12
55. Kaur A, Dhiman G (2019) A review on search-based tools and techniques to identify bad code smells in object-oriented systems. Adv Intell Syst Comput 741:909–921. https://doi.org/10.1007/978-981-13-0761-4_86
56. Singh S, Kaur S (2017) A systematic literature review: refactoring for disclosing code smells in object oriented software. Ain Shams Eng J 9(4):2129–2151. https://doi.org/10.1016/J.ASEJ.2017.03.002
57. Li L et al (2017) Static analysis of android apps: a systematic literature review. Inform Softw Technol 88:67–95. https://doi.org/10.1016/j.infsof.2017.04.001
58. Degu A (2019) Android application memory and energy performance: systematic literature review. IOSR J Comp Eng 21(3):20–32
59. Qiu L, Wang Y, Rubin J (2018) Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis - ISSTA, pp 176–186, doi:https://doi.org/10.1145/3213846.3213873

60. Corrodi C, Spring T, Ghafari M, Nierstrasz O (2018) Idea: benchmarking android data leak detection tools. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Jun 2018, vol 10953 LNCS, pp 116–123, doi: https://doi.org/10.1007/978-3-319-94496-8_9
61. Ndagi JY, Alhassan JK (2019) Machine learning classification algorithms for adware in android devices: a comparative evaluation and analysis. In: Proceedings of the 15th international conference on electronics, computing, and computation - ICECCO, Dec 2019, pp 1–6, doi: https://doi.org/10.1109/ICECCO48375.2019.9043288
62. Cooper VN, Shahriar H, Haddad HM (2014) A survey of android malware and mitigation techniques. In: Proceedings of the 11th international conference on information technology: new generations, Apr 2014, pp 327–332, doi: https://doi.org/10.1109/ITNG.2014.71
63. Li L, Bissyande TF, Klein J (2019) Rebooting research on detecting repackaged android apps: literature review and benchmark. IEEE Trans Softw Eng:1–1. https://doi.org/10.1109/tse.2019.2901679
64. Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci Comput Program 74(7):470–495. https://doi.org/10.1016/j.scico.2009.02.007
65. Wang Y, Li Y, Lan T (2017) Capitalizing on the promise of Ad prefetching in real-world mobile systems. In: Proceedings of the IEEE 14th international conference on mobile Ad Hoc and sensor systems - MASS, Oct 2017, pp 162–170, doi:https://doi.org/10.1109/MASS.2017.46
66. Rasmussen K, Wilson A, Hindle A (2014) Green mining: energy consumption of advertisement blocking methods. In: Proceedings of the 3rd international workshop on green and sustainable software - GREENS, pp 38–45, doi:https://doi.org/10.1145/2593743.2593749
67. Shao Y, Wang R, Chen X, Azab AM, Mao ZM (2019) A lightweight framework for fine-grained lifecycle control of android applications. In: Proceedings of the 14th EuroSys conference - EuroSys, pp 1–14, doi:https://doi.org/10.1145/3302424.3303956
68. Petersen K, Feldt R, Mujtaba S, Mattsson M (2008) Systematic mapping studies in software engineering. In: Proceedings of the 12th international conference on evaluation and assessment in software engineering - EASE, pp 68–77
69. Fowler M (2002) Refactoring: improving the design of existing code. In: Extreme programming and agile methods — XP/Agile universe. Springer, Berlin, pp 256–256
70. Pathak A, Charlie Hu Y, Zhang M (2011) Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In: Proceedings of the 10th ACM workshop on hot topics in networks (HotNets-X). Association for Computing Machinery, New York, NY, Article 5, 1–6. doi:https://doi.org/10.1145/2070562.2070567
71. Yasumatsu T, Watanabe T, Kanei F, Shioji E, Akiyama M, Mori T (2019) Understanding the responsiveness of mobile app developers to software library updates. In: Proceedings of the 9th ACM conference on data and application security and privacy - CODASPY, pp 13–24, doi: https://doi.org/10.1145/3292006.3300020
72. Alrubaye H, Mkaouer MW, Khokhlov I, Reznik L, Ouni A, Mcgoff J (2020) Learning to recommend third-party library migration opportunities at the API level. Appl Soft Comput 90:106140. https://doi.org/10.1016/j.asoc.2020.106140
73. Nguyen PT, Di Rocco J, Di Ruscio D, Di Penta M (2020) CrossRec: supporting software developers by recommending third-party libraries. J Syst Softw 161:110460. https://doi.org/10.1016/j.jss.2019.110460
74. Ouni A, Kula RG, Kessentini M, Ishio T, German DM, Inoue K (2017) Search-based software library recommendation using multi-objective optimization. Inf Softw Technol 83:55–75. https://doi.org/10.1016/j.infsof.2016.11.007
75. Saied MA, Ouni A, Sahraoui H, Kula RG, Inoue K, Lo D (2018) Improving reusability of software libraries through usage pattern mining. J Syst Softw 145:164–179. https://doi.org/10.1016/j.jss.2018.08.032