Tool Support for Green Android Development: A Systematic Mapping Study

Iffat Fatima[1], Hina Anwar[2], Dietmar Pfahl[2] and Usman Qamar[1]
[1]*College of Electrical and Mechanical Engineering, National University of Sciences and Technology, Islamabad, Pakistan*
[2]*Institute of Computer Science, University of Tartu, Tartu, Estonia*

Keywords: Android Development, Code Refactoring, Energy Efficient, Green Software Engineering, Mapping Study.

Abstract: In order to make mobile apps energy efficient, we must find ways to support energy efficient app development. While there is a lack of support tools that aid practitioners in moving towards green Android development. Our goal is to establish the state of the art with respect to support tools that aid green Android development and to identify opportunities for further research. To achieve this goal, we conduct a systematic mapping study. After applying inclusion, exclusion and quality criteria we selected 21 studies for further analysis. Current support tools to aid green Android development were classified into three categories: Profiler, Detector and Optimizer. Most Profiler tools provide a graphical representation of energy consumed over time at various levels. 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 generate refactored version(s) of APK/SC. The most typical technique used by Detector and Optimizer tools is static source code analysis using a predefined set of rules. Profiler tools use a wide range of techniques to measure energy consumption. However, these tools have limitations in terms of code smell/energy bug coverage, accuracy, and usability.

1 INTRODUCTION

With the growing awareness of global warming and its causes related to human activities (Powell, 2019), the information and communication technology (ICT) as well as software engineering (SE) research communities show growing interest in green software development. Software systems have such a significant impact on our everyday lives that changes towards environmental sustainability can ripple to other systems with which they interact and positively affect the industries in which they are used (Anwar and Pfahl, 2017). It was reported that the use of ICT can eliminate 12.1 billion tons of CO^2 emissions per sector per year till 2030 (GeSI, 2015). While these estimates are promising, the current ICT and software development practices are causing harm to the environment. There is ample research on making the hardware energy efficient but research on making energy efficient software is still emerging. With emerging technological trends such as internet of things (IoT), the importance of portable devices like smartphones is imperative. There were over 1.9 billion smartphones sold in 2018 alone (Gartner, Inc., 2018b) and the numbers are increasing every year. According to an online survey users spend most of their digital media time on smartphones using mobile apps(Mindsea, 2019). In the smartphone market, Android has the biggest market share globally, i.e., 87.8% (Gartner, Inc., 2018a). Portable devices like smartphones have limited hardware resources such as battery and memory. Therefore, Android apps need to be developed with energy efficiency in mind. In this paper, we use the term green Android development for research that is intended to help developers design and implement energy efficient Android apps. Recent studies focus on novel energy profiling/enhancement tools and methods in mobile apps (Ardito et al., 2013; Banerjee and Roychoudhury, 2016; Chung et al., 2011; Fernandes et al., 2014; Kansal and Zhao, 2008; Pathak et al., 2012). However, there is still a lack of information, support infrastructure and tools for developers to make energy efficient applications (Anwar and Pfahl, 2017; Manotas et al., 2016). To better understand what is available and what is lacking with regards to support tools, we conducted a systematic mapping study. Our results indicate that there are many different support tools developed to help aid developers at various levels, however, they are not open source and require advance knowledge related to energy related

Fatima, I., Anwar, H., Pfahl, D. and Qamar, U.
Tool Support for Green Android Development: A Systematic Mapping Study.
DOI: 10.5220/0009770304090417
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 409-417
ISBN: 978-989-758-443-5
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

issues in Android apps. There is a need to develop a consolidated support tool that covers all the energy related issues and is simpler to integrate with current integrated development environments (IDEs).

2 RELATED WORK

Secondary studies related to tool support for improving energy efficiency in Android development are scarce. Some secondary studies (Fontana et al., 2011; Kaur and Dhiman, 2019; Singh and Kaur, 2018) reviewed tools and techniques for improving the quality of Java projects in the object-oriented paradigm (with regards 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 (Fontana et al., 2011; Kaur and Dhiman, 2019; Singh and Kaur, 2018) 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. (Li et al., 2017) 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 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 (Degu, 2019) performed a systematic literature review to classify primary studies with a focus on resource usage, energy consumption, and performance in Android apps. The author has reviewed work published between 2012 and 2017. The results did not provide an in depth review of support tools in green Android development. 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 secondary studies discussed above have covered published work until 2015 and many of the reviewed tools in those studies are now outdated/obsolete. Therefore, in this paper, we provide a different view of literature by analyzing recently developed support tools for energy profiling, code optimization and refactoring in Android development to improve energy efficiency.

3 RESEARCH METHODOLOGY

In order to be able to develop better support tools we need to understand what is already available, what is still needed, and how the problems in existing tools could be overcome. Following the method described in (Petersen et al., 2008) we formulated research questions, then based on those research questions we formulated a general search query and conducted the search in the following online repositories for primary studies: IEEE Xplore, ACM digital library, Science Direct, and Springer. In this paper, we cover studies from 2014 to June 2019, as from 2014 onwards the focus of many studies has been Android and energy efficient app development, indicating a shift in research focus.

3.1 Research Questions

We formulated the following research question with three sub-questions.
RQ1: *What state of the art support tools have been developed to aid software practitioners in developing energy efficient Android apps?*
RQ1.1: *How do existing support tools compare to one another in terms of the support they offer to practitioners for improving energy efficiency?*
RQ1.2: *How do existing support tools compare to one another in terms of techniques they use for offering the support?*
RQ1.3: *What are the limitations of existing support tools developed to help aid software practitioners in developing energy efficient Android apps?*

3.2 Search Query

We derived search terms from the research goals and research questions. We looked for alternatives and synonyms of the search terms in studies we already knew and refined our search terms to return the most relevant studies. We used the * operator to cover possible variations of the selected search terms in the search query. Keyword OR was used to improve search coverage. The AND NOT keyword was used to eliminate papers that do not fall into the scope of this study e.g. environmental, education and hardware related papers that discuss energy efficiency solutions. We did not use the search terms 'mobile development', 'apps', 'sustainability', 'green', 'optimization', 'recommendation', and 'energy' as they were too high level and produced a larger corpus consisting of higher number of irrelevant studies. The search terms 'resource leak', 'tool', 'framework', and 'technique' were eliminated to avoid being too specific. The search query was applied to popular online repositories (IEEE Xplore, ACM digital library, Science Direct, and Springer). In each repository, based on available advanced search options, filters were ap-

Table 1: Search query filter.

Sr.	Filter	Value
1	Publication year	≥ 2014
2	Content Format	PDF
3	Content Type	Journal Article, Conference Paper
4	Publication Area	Software Engineering

plied to refine the query results. Applied filters are shown in Table 1. The search query was applied to the titles, abstracts, and keywords of the papers.
Android develop AND (energy effici* OR energy bugs OR code optimization OR code refactoring) AND NOT (testing OR environment* OR edu* OR hardware).*

3.3 Screening of Papers

We removed duplicate papers and then defined inclusion, exclusion, and quality criteria for further screening of search results. The search results from online repositories were first loaded in Zotero[1] (an open source reference management system) to create a dataset of relevant studies. Using the feature in Zotero duplicates studies were removed from the dataset. Next, we manually applied inclusion, exclusion (see Table 2) and quality criteria (see Table 3) on the remaining papers. Abstracts of the papers and structure of the paper were assessed for further quality assessment. A maximum quality score of 3 could be assigned to a paper. If a paper was below a quality score of 1.5 it was removed from the results.

3.4 Classification and Data Extraction

To answer RQ1 and its sub-questions, we identified the main keywords of the selected studies along with the commonly used terms in the abstract to define categories of support tools. Research methodology and results of selected studies were additionally studied when needed. We kept extracted data in excel spread sheets for further processing. During data extraction, if there was a conflict of opinion it was discussed among authors of this paper until a consensus was reached. To answer RQ1, a bottom-up merging technique was adopted to build our own classification scheme (see Table 4). Once a classification scheme was established, we extracted data from each selected study to identify its main contribution and assigned the tool mentioned in the study to a category based on the classification scheme.

To answer RQ1.1, we extracted data form each selected study to gather information about the kind of support the identified tool offers based on the inputs of the tool, outputs of the tool, recommendation(s)

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

Table 2: Inclusion/Exclusion criteria.

Inclusion	Exclusion
Primary studies should be related to software engineering with a focus on energy efficiency in Android apps. A tool/automated technique for code optimization and refactoring or energy profiling was presented for green Android development. We considered only conference and journal articles for which full text was available.	Studies focused on energy efficiency with respect to hardware or environmental issues or present secondary data were excluded. The work presented in a thesis or book chapter is usually published in relevant journals or conferences as well. Therefore, theses, magazine articles, book chapters, work-in-progress papers, and papers that were not in English were also excluded.

Table 3: Quality assessment criteria.

ID	Description	Rating
1	Does the paper clearly state contribution(s) that is directly related to improving energy efficiency in Android apps?	1
2	Is the research method adequately explained?	1
3	Are threats to validity and future research directions separately discussed?	1
	TOTAL	3

Table 4: Categories of support tools.

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.

made by authors based on the output of the tool, code smells/energy bugs and measurement covered by the tool. In general, a code smell is defined as a surface indication that usually corresponds to a deeper problem in the system (Fowler, 2002) and an energy bug is defined as 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 (Pathak et al., 2011). In the light of these definitions, we looked for Android specific code smells and energy bugs in the studies.

To answer RQ1.2, a classification scheme was needed to classify techniques used in support tools for improving the energy efficiency of apps. 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 Table 5. Once we had established the classification scheme, we extracted data from the abstract and research methodology of each selected study and assigned it to a category defined in the classification scheme. To answer RQ1.3, we studied the results, threats to validity and conclusion of selected studies and summarized the

limitation of support tools in each category.

4 RESULTS

4.1 Screening

As a result of running the search query and applying filters (see Table 1) on search results, 1462 studies were found from the selected online repositories. These studies were loaded into the Zotero software for the screening and removal of duplicates, the total number of papers were reduced to 1377 after duplicate removal. Inclusion and exclusion criteria were applied to the remaining studies and the number was reduced to 566. We read abstracts of these studies and looked at the paper structure to assign them a quality score based on quality criteria. After applying the quality criteria, the number of selected studies[2] was reduced to 21. (See Tables 6, 7, and 8)

4.2 Classification and Analysis

In this section, we present the result[2] for RQ1 and its sub-questions RQ1.1 to RQ1.3.

RQ1: *What state of the art support tools have been developed to aid software practitioners in developing energy efficient Android apps?*

To answer RQ1, the classification scheme defined in Section 3.4 was adopted and the selected studies were divided into categories: 1) Profiler, 2) Detector, 3) Optimizer, based on the support tool they offer to aid green Android development. Table 9 gives an overview of the distribution of selected studies in each category along with the total number of tools in each category. Figure 1 shows the number of publications each year. The color in bars indicates the number of tools in each category each year. From 2014 to 2017 we can see a decrease in the number of Profiler tools while an increase in the number of Optimizer tools. At least one Detector tool has been published every year from 2014 to 2018. In 2019 (till June), no new support tool has been published.

RQ1.1: *How do existing support tools compare to one another in terms of the support they offer to practitioners for improving energy efficiency?*

To answer RQ1.1, we provide a list of all the tools identified in each category. In Table 10[2] the column input provides information about what is the input for each tool. The column output provides information about the support the tool offers based on the input.

[2]For additional information, e.g., list of selected studies, additional tables and figures, see URL: https://figshare.com/s/7e78f2469727c31d2957

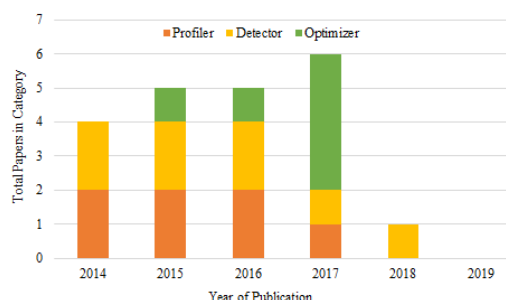

Figure 1: Number of publications per year per category.

Studies in the category Profiler offer support to the practitioners by providing tools that can measure the energy consumed by 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. In some studies, authors further discuss how the output produced from their tool could be used to improve the energy consumption of apps. We present this information in Table 11 as a list of recommendations provided by those studies in the Detector category along with a reference to the selected study. We define recommendation as; a fix for an energy bug/code smell that can be applied in a finite number of steps/actions in a specific context.

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.

RQ1.2: *How do existing support tools compare to one another in terms of techniques they use for offering the support?*

To answer RQ1.2 we discuss in detail techniques used in each tool for improving the energy efficiency of apps. Table 12 gives an overview of tools and techniques along with the reference to selected studies. 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 Section 3.4).

Profiler. Profiling tools measure the energy consumption of the software and the used hardware re-

Table 5: Categories of techniques used for providing support to aid green Android development.

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 find out 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 consumption.
T3	Logcat Analysis	A technique that uses system level log files to obtain energy consumption information provided by 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 analyses it using one or 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 run time for the detection of vulnerabilities in the app under test. It monitors the spread of sensory data during different app states.

Table 6: Number of studies extracted per online repository.

Sr.	Repo.	# of papers	Conference Papers	Journal Articles
1	IEEE Xplore	832	621	211
2	ACM Digital library	478	454	24
3	Springer	130	95	35
4	Science Direct	22	4	18

Table 7: Number of articles per screening step.

Sr.	Step in the screening of papers	# of papers
1	Search string results after applying filters	1462
2	Remove duplicates	1377
3	Apply inclusion and exclusion criteria	566
4	Apply quality criteria	21

Table 8: Quality score assigned to each selected study.

Study ID	Quality Score
S3, S6, S15, S18, S13	2.5
S1, S2, S4, S5, S8, S9, S10, S16, S19	2
S11, S12, S14, S17, S20, S21, S7	1.5

Table 9: Distribution of studies in each category.

ID	Selected Studies	# Tools
CP	S6, S14, S16, S12, S13, S20, S19	7
CD	S1, S3, S4,S5, S8, S9,S7, S17	8
CO	S10, S11, S15, S18 S2, S21	6

sources. Profilers [S6, S16, and S14] were designed to inject logging statements into the Smali files to gather relevant log data. This data was then transformed as needed and matched against pre-defined API names or method names to identify energy intensive APIs and methods in the program. In these profilers, Dumpsys or System level logs or ADB data plotted on a graph was used to measure energy usage. Another group of profilers analyzed the paths in source codes using finite state models (FSM) or inter procedural control-flow paths. The authors of [S19] created a dynamic model that instrumented the app to measure its power consumption. Methods were invoked using tests and were classified into three different levels (classes, packages, and projects) based on their energy consumption. The classification was then presented graphically. The authors of [S12] modeled energy consumption as an FSM model by collecting

Table 10: List of support tools and their inputs and outputs.

Ct.	Tool	Input	Output	ID
	Orka	APK	ECG	S6
	SEPIA	AE	ECG	S12
	Mantis	PBC	Program CRC predictors	S13
CP	AEP	SL, PID via ADB	ECG	S14
	E-Spector	SL, AL via ADB	ECG	S16
	SEMA	PID, MVC	Log of Energy consumption	S20
	Keong et. al	SC	ECG	S19
	Wu et al.	SC	List of energy bugs	S1
	Kim et al.	PBC	List of energy bugs	S3
	Statedroid	APK	List of energy bugs	S5
CD	PatBugs	SC	List of detected warnings	S8
	SAAD	APK	List of energy bugs	S9
	aDoctor	SC	List of code smells	S4
	GreenDroid	PBC,CF	List of energy bugs + severity level	S17
	Paprika	APK, PM	List of code smells	S7
	DelayDroid	APK	Refactored APK	S2
	HOT-PEPPER	APK	Most energy efficient APK, Refactored SC, and List of refactoring	S10
CO	Asyncdroid	SC	Refactored SC	S11
	EARMO	APK	Refactored APK	S15
	EnergyPatch	APK	Refactored APK	S18
	Nguyen et al.	SC	Refactored SC	S21

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, PM=PlayStore Metadata.

Table 11: Recommendation given by authors of studies in 'Detector' category.

Category: Detector	
ID	Recommendation
S1	Energy bug (related to listener leaks) caused due to control flow issues in apps could be fixed by adding and removing interface listeners and correct handling of Android lifecycle callbacks.
S3	Energy bugs (related to texture transfers, image and frame rending) in graphic intensive apps could be fixed in three ways: (1) Loop invariant graphic textures should be moved out of the loop. (2) Group related images together for efficient rendering. (3) Identify identical frames to prevent redundant frame drawing.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 12: Overview of support tools showing the technique used for offering support to developers.

		Techniques						
Ct.	Tool	T1	T2	T3	T4	T5	T6	ID
CP	Orka	✓						S6
	SEPIA		✓					S12
	Mantis		✓					S13
	AEP			✓				S14
	E-Spector	✓						S16
	SEMA			✓				S20
	Keong et. al						✓	S19
CD	Wu et al.				✓			S1
	Kim et al.				✓			S3
	Statedroid				✓			S5
	PatBugs				✓			S8
	SAAD				✓			S9
	GreenDroid						✓	S17
	Paprika				✓			S7
	aDoctor							S4
CO	DelayDroid				✓			S2
	HOT-PEPPER				✓			S10
	Asyncdroid				✓			S11
	EARMO					✓		S15
	EnergyPatch						✓	S18
	Nguyen et al.				✓			S21

events from an Android device. By executing the instrumented unit tests the line of source code with the energy consumption issue was identified. The authors of [S13] sliced the program into executable slices for generating feature predictors. Predictors were generated based on some predefined metrics which quantified energy consumption. The authors of [S20] measured energy consumption by encapsulating the code blocks with energy evaluation functions. Sampling collector extracted energy consumption and a runtime manager extracted application information from OS.

Detector. The authors of [S17] used JPF to statically analyze the state space of the app to find the usage of sensors and wake locks. The utilization of sensory data was defined by a coefficient. The authors of [S7] converted APK files into byte code using SOOT which is a static source code analyzer. The converted code was then used to identify and report various anti-patterns in code. The authors of [S5] identified resource intensive items in an app using control-flow graphs (CFGs). Using resource protocols as a guide a taint-like analysis was performed on CFGs. In the study [S1] valid inter-procedural control-flow paths were analyzed in each callback method and its transitive callees, in order to detect energy draining operations related to missing deactivation behaviors. In the study [S3], a static analysis tool SOOT was used to find energy bugs in graphics intensive mobile apps along with point-to analysis tool, SPARK. Energy bugs were identified based on the frequency at which GPU was being used to perform certain actions e.g. texture transformations etc. The authors of [S8] used flexible bug pattern specification notation (FBPSN) to specify bug patterns. The source code was transformed to CFGs which were used to detect bugs with the help of FBPSN specified bug patterns. In the study [S9] the APK files were analyzed using APKTool and resource leak were analyzed using SAAF. Lint was used for layout defect analysis. 'SAAF' internally uses inter-procedural, intra-procedural and component call analysis and resource leak detection. The output from 'Lint' was passed through a filter based on a set of rules defined in a defect table to generate the report. The results of both these tools are used to generate the respective reports. In the study [S4] an abstract syntax tree of classes was traversed to apply a set of detection rules based on a specific set of code smells.

Optimizer. The authors of [S2] introduced the tool DelayDroid which used static analysis and byte code refactoring using ASM library to find the parts of code that performed energy intensive network related tasks and batched them together to perform those tasks at a delayed interval to reduce energy consumption. The tool Hot-pepper was presented in [S10] which detected energy smells using the Paprika tool and computed the energy consumption of code smells using the Naga-Viper tool. Corrections were made using a tool called SPOON that used static analysis for transformation. The authors of [S11] presented a tool developed on top of the Eclipse refactoring engine which converted AsyncTask to IntentService in order to improve the asynchronous operations. In the study [S15] a novel tool EARMO was presented which created code abstractions to search for anti-patterns based on QMOOD metrics. To correct anti-patterns in the app, the ReCon tool was used. In [S18] a framework was presented that had three main components: Design extractor, refactoring component and code generation component. Design and defect expressions were generated from EFG using deterministic finite automata. Their intersection was

Table 13: Android energy bugs detected by each tool in 'Detector' and 'Optimizer' categories.

Tool	RL	WB	VBS	IB	TMV	TDL	NCD	UL	UP	Ref
Wu et al.	✓									S1
GreenDroid	✓	✓								S17
Kim et al.							✓			S3
Statedroid	✓	✓	✓							S5
PatBugs										S8
SAAD	✓				✓	✓	✓	✓	✓	S9
Paprika	✓						✓			S7
aDoctor	✓	✓								S4
DelayDroid			✓							S2
HOT-PEPPER	✓						✓			S10
Asyncdroid			✓							S11
EARMO										S15
EnergyPatch	✓	✓	✓	✓						S18
Nguyen et al.										S21

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

Table 14: Android code smells detected by each tool in the 'Detector' and 'Optimizer' categories.

Tool	Ref	DTWC	DR	IDFP	IDS	ISQLQ	IGS	LIC	LT	MIM	NLMR	PD	RAM	SL	UC	LC	LWS	UHA	BFU	UIO	IWR	HAT	HSS	HBR	IOD	ERB
Wu et al.	S1														✓	✓	✓									
Kim et al.	S3																								✓	
Statedroid	S5														✓	✓										
PatBugs	S8														✓											
SAAD	S9																									
ADoctor	S4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓											
Paprika	S7				✓		✓	✓		✓	✓							✓	✓	✓	✓	✓	✓	✓	✓	
GreenDroid	S17														✓											
DelayDroid	S2																									
HOTPEPPER	S10				✓		✓	✓		✓	✓							✓	✓	✓	✓	✓	✓	✓	✓	
AsyncDroid	S11																					✓				
EARMO	S15				✓		✓																			✓
EnergyPatch	S18														✓											
Nguyen et al.	S21								✓					✓												

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

used to refactor the code. The authors of [S21] used PMD and Android Lint to create an Eclipse plugin for refactoring of source code. PMD created an Abstract Source Tree (AST) to analyze code and apply the predefined rules. Table 13 lists Android energy bugs[2] that are covered by tools in the Detector and Optimizer categories. Android energy bugs[2] TMV, TDL, UL, UP are detected by 10% of the tools, whereas RL is detected by 80% of tools, in the Detector category. However, none of the tools in the Optimizer category covers them. RL and VBS energy bugs are detected by 30% and 50% of the tools in the Optimizer category, respectively. Table 14 shows which Android code smells[2] are covered by which individual tools in Detector and Optimizer categories. Android code smell[2] ERB is not detected by any tool in the Detector category but 17% of tools in the Optimizer category cover them. Android code smells such as LWS, LC, RAM, PD, ISQLQ, IDFP, DR, and DTWC are detected by 13-25% of the tools in Detectors category but none of the tools in Optimizer category can identify these smells.

RQ1.3: *What are the limitations of existing support tools developed to help aid software practitioners in developing energy efficient Android apps?*

Profiler. Tools included in this category had several limitations. Energy measurements were less accurate as compared to measurements taken with an external device. The error rate was high. Limitations on what could be profiled e.g. energy consumption of only some parts of the app were profiled or only framework or only app or only kernel level profiling was provided. The impact of techniques like code injections and extensive logging came with an execution overhead and might have affected the final energy costs. Resources used by software based profiler might not be accounted for in the energy costs. Only a limited number and formats of input were handled by these profilers. Model based energy profiling approaches might ignore the run time information about apps.

Detector. Tools included in this category had several limitations. As most of the tools in this category were using static source code analysis techniques to offer support, they might produce some false positives/negatives. Some of these tools used open source static source code analyzer, which could affect their results and accuracy. A limited number of Android code smells/energy bugs were identified using each tool and most of the tools did not cover multi-threaded programs and multiple interacting objects.

Optimizer. Tools in this category had several limitations. Refactored APK/SC generated by the tools did not offer acceptable trade off in most cases with performance, usability or other non-functional requirements. The refactoring done by tools needed manual verifications. Only a limited number of Android code smells/ energy bugs were automatically identified and corrected by these tools. Manually applying the refactorings suggested by these tools required expertise in programming and energy related issues. Many tools in this category were also dependent on static source code analysis techniques therefore, they might produce false positives/ negatives.

5 DISCUSSION

Based on the results of RQ1 and its sub-question RQ1.1 we observed that the selected studies focus on a specific niche of code smells, energy bugs and software metrics. There is a need for generic support tools that can cover the functionalities of all three categories of tools discussed in the results section. From the results of RQ1.2, the support tools in Profiler category used many different techniques. The tools in Detector and Optimizer categories frequently used static source code analysis technique, which might result in producing false positive and false negatives. In all categories, none of the tools use a combination of techniques. Static source code analysis techniques do not cover dynamic issues e.g. issues related to Async. Therefore, hybrid techniques need to be used for the development of new tools using a combination of static and dynamic analysis. For the development of better support tools, the evaluation of the tools on industrial projects should be increased. Most of the current support tools were only evaluated using open source data sets and there are indications (see related work) that industrial development is different. Therefore, the industry relevance of the support tools might not be obvious because they are not evaluated in industrial settings. Software based metrics that affect energy consumption need to be clearly identified and correlated to improve code smell detection. Results from studies that focus on correlating the structure of source code to software metrics could be used for this purpose. In addition, non-intrusive techniques should be used to collect metrics. Based on the results of RQ1.3, the current state of the art tools could be extended into complete frameworks by combining them with one another or combining them with other industrially famous code analyzers like Android Lint, Check Style, Find Bugs and PMD. The results from the selected studies 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.
From the discussion, we list desired functionalities in the new breed of support tools to aid green Android development:

- Must be integratable with the Android Studio.
- Must have full coverage of Android specific energy bugs.
- Must be able to handle multi-threaded programs.
- Must be able to handle multi interacting objects.
- Must have an interface consistent with the Android studio to increase usability.
- Must have elaborated help and documentation available offline for every function and command.
- Must have complete coverage of object oriented (such as all 22 Fowler) code smells
- Must have complete coverage of Android specific code smell.
- Must have the capability of profiling energy consumption of app at all levels such as method level, class level, package level, and Kernel level.
- The energy level must be represented in graphs.
- Energy graphs must be linked to source code. I.e. from graph user could be directed to the codes causing a change in energy consumption.
- Must be able to handle both static and dynamic analysis of source code.
- Must offer trade-offs with other non-functional requirements in the form of recommendations.
- Must provide a preview of refactoring/optimization choices on the code before applying them.

6 THREATS TO VALIDITY

The search query and classification of selected studies could be biased by the researchers knowledge. We mitigated this threat by defining the inclusion, exclusion and quality criteria for the selection of the studies. Conflicting opinions were discussed among authors of this paper until a consensus was reached. In order to avoid false positive and false negatives in the search results, we used the wildcard character (*) to maximize coverage and the keyword AND NOT to remove irrelevant studies. The results of the search strings were manually checked and further refined by the authors. Online repositories continuously update their databases to include new publication, therefore, executing the same query might yield some additional results that were not included in this study. We already knew about many relevant studies and we recaptured almost 85% of them when we executed the search query. On each online repository the search mechanism is slightly different we tried to keep the query as consistent as possible, but there might be slight difference due to the difference in search mechanism provided by different online repositories. Some selected studies use the terms code smells and energy bugs interchangeably which could affect the classification. To mitigate this threat we used the selected definitions (c.f section 3.4) for code smells and energy bugs to correctly classify the studies in the right category. We focused particularly on the support tools

for energy profiling, code optimization and refactoring of code smells/energy bugs to help aid green Android development. Other types of support tools, such as tools for interface optimization, third-party library detection etc., were not in the scope of this study. We plan to cover such tools in future work.

7 CONCLUSION

To get an overview of the state of the art and to find research opportunities with respect to support tools available for green Android development, we conducted a mapping study. Based on our analysis the current support tools were classified into three categories 1) Profiler, 2) Detector, 3) Optimizer. The main findings of the paper 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 automatically convert original APK/SC to a refactored version(s) of APK/SC. The most typical technique in Detector and Optimizer category was static source code analysis using a predefined set of code smells and rules.

ACKNOWLEDGEMENTS

This work is supported by the Estonian Center of Excellence in ICT research (EXCITE), the group grant PRG887 funded by the Estonian Research Council.

REFERENCES

Anwar, H. and Pfahl, D. (2017). Towards greener software engineering using software analytics: A systematic mapping. In *Proc. of the 43rd Euromicro Conf. on Soft. Eng. and Advanced Applications*. IEEE.

Ardito, L., Procaccianti, G., Torchiano, M., and Migliore, G. (2013). Profiling power consumption on mobile devices. In *Proc. of the 3rd Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Tech.*, pages 101–106.

Banerjee, A. and Roychoudhury, A. (2016). Automated re-factoring of android apps to enhance energy-efficiency. In *Proc. of the Int. Workshop on Mobile Soft. Eng. and Sys.* ACM Press.

Chung, Y.-F., Lin, C.-Y., and King, C.-T. (2011). ANEPROF: Energy profiling for android java virtual machine and applications. In *Proc. of the 17th Int. Conf. on Parallel and Distributed Sys.* IEEE.

Degu, A. (2019). Android app memory and energy performance: Systematic literature review. *IOSR J. of Comp. Eng.*, 21.

Fernandes, T. S., Cota, E., and Moreira, A. F. (2014). Performance evaluation of android applications: A case study. In *Proc. of the Brazilian Symp. on Computing Sys. Eng.* IEEE.

Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., and Tonello, A. (2011). An experience report on using code smells detection tools. In *Proc. of the 4th Int. Conf. on Soft. Testing, Verification and Validation Workshops*. IEEE.

Fowler, M. (2002). Refactoring: Improving the design of existing code. In *Extreme Programming and Agile Methods — XP/Agile Universe*, pages 256–256. Springer Berlin Heidelberg.

Gartner, Inc. (2018a). Gartner says huawei secured no. 2 worldwide smartphone vendor spot, surpassing apple in second quarter 2018. Accessed: 2019-08-30.

Gartner, Inc. (2018b). Gartner says worldwide end-user device spending set to increase 7 percent in 2018; global device shipments are forecast to return to growth. Accessed: 2019-08-30.

GeSI (2015). Smarter2030 ict solutions for 21st century challenges. Technical report.

Kansal, A. and Zhao, F. (2008). Fine-grained energy profiling for power-aware application design. *ACM Performance Evaluation Review*, 36:26–31.

Kaur, A. and Dhiman, G. (2019). A review on search-based tools and techniques to identify bad code smells in object-oriented systems. In *Harmony Search and Nature Inspired Optimization Algo.*, pages 909–921. Springer Singapore.

Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. (2017). Static analysis of android apps: A systematic literature review. *Info. and Soft. Tech.*, 88:67–95.

Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., and Clause, J. (2016). An empirical study of practitioners' perspectives on green software engineering. In *Proc. of the 38th Int. Conf. on Soft. Eng.* ACM Press.

Mindsea (2019). 25 Mobile App Usage Statistics To Know In 2019. Accessed: 2019-08-05.

Pathak, A., Hu, Y. C., and Zhang, M. (2011). Bootstrapping energy debugging on smartphones. In *Proc. of the 10th ACM Workshop on Hot Topics in Networks*. ACM Press.

Pathak, A., Hu, Y. C., and Zhang, M. (2012). Where is the energy spent inside my app? In *Proc. of the 7th ACM european conf. on Computer Sys.* ACM Press.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic mapping studies in soft. eng. BCS Learning & Development.

Powell, J. (2019). Scientists reach 100 *Bulletin of Science, Tech. and Society*, 37:183–184.

Singh, S. and Kaur, S. (2018). A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Eng. J.*, 9(4):2129–2151.